

RE

AD-A278 036 ON PAGE

Form Approved  
OPM No.

Public reporting burden  
and maintaining the  
suggestions for reducing  
22202-4302, and to the Office of Management and Budget, Washington, DC 20503.



response, including the time for reviewing instructions, searching existing data sources gathering  
its regarding this burden estimate or any other aspect of this collection of information, including  
Information Operations and Reports: 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA  
and Budget, Washington, DC 20503.

1. AGENCY USE

(Leave)

2. REPORT

3. REPORT TYPE AND DATES

4. TITLE AND

5. FUNDING

- TLD Systems, Ltd. / VC# 940 308 W-11335  
- TLD Conanche VAX/1960 Ada Compiler Systems  
Version 4.1.1

6.

Authors:

Wright-Patterson AFB

7. PERFORMING ORGANIZATION NAME(S) AND

8. PERFORMING  
ORGANIZATION

Ada Validation Facility Language Control Facility/  
ASD/SCIL 2513g-676, Room 135

Wright-Patterson AFB, Dayton OH 45433

9. SPONSORING/MONITORING AGENCY NAME(S) AND

10. SPONSORING/MONITORING  
AGENCY

Ada Joint Program Office  
The Pentagon, Rm 3E118  
Washington, DC 20301-3080

DTIC

ELECTE

APR 12 1994

11. SUPPLEMENTARY

12a. DISTRIBUTION/AVAILABILITY

12b. DISTRIBUTION

This document has been approved  
for public release and sale; its  
distribution is unlimited.

13.

(Maximum 200)

Host: DEC Local Area Network VAX cluster (comprising  
main VAX 3100 model 90 machines) (under VMS 5.3a)

Target: Tronix JIAWG Execution Vehicle (1960mx)  
bare machine, using TLD RealTime Executive  
(TLD rtx) (Domain Configuration), Version 4.1.1

14. SUBJECT

Ada programming language, Ada Compiler Validation Summary  
Report, Ada Compiler Val. Capability Val. Testing, Ada Val.  
Office: Ad Val. Facility ANS-MIL-STD-1546 A LSP

15. NUMBER OF

16. PRICE

17. SECURITY

CLASSIFICATION  
UNCLASSIFIED

18. SECURITY

UNCLASSIFIED

19. SECURITY

CLASSIFICATION  
UNCLASSIFIED

20. LIMITATION OF

UNCLASSIFIED

NSN

Standard Form 298, (Rev. 2-89)  
Prescribed by ANSI Std.

AVF Control Number: AVF-VSR-582.0394  
Date VSR Completed: March 14, 1994  
94-02-14-TLD

Ada COMPILER  
VALIDATION SUMMARY REPORT:  
Certificate Number: 940305W1.11335  
TLD Systems, Ltd.

TLD Comanche VAX/i960 Ada Compiler System, Version 4.1.1  
VAX Cluster under VMS 5.5 =>  
Tronix JIAWG Execution Vehicle (i960MX)  
under TLD Real Time Executive, Version 4.1.1

(Final)

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

Prepared By:  
Ada Validation Facility  
645 CCSG/SCSL  
Wright-Patterson AFB OH 45433-5707

DTIC QUALITY INSPECTED 3

94-11000



9400

94 4 11 11 2

## Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 5 March 1994.

Compiler Name and Version: TLD Comanche VAX/i960 Ada Compiler System,  
Version 4.1.1

Host Computer System: DEC Local Area Network VAX Cluster (comprising  
2 MicroVAX 3100 Model 90 machines) (VMS 5.5)

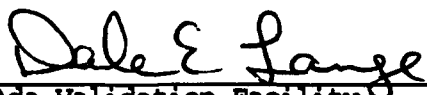
Target Computer System: Tronix JIAWG Execution Vehicle (i960MX)  
under TLD Real Time Executive (TLDrtx)  
(Domain Configuration), Version 4.1.1

Customer Agreement Number: 94-02-14-TLD

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 940305W1.11335 is awarded to TLD Systems, Ltd. This certificate expires two years after MIL-STD-1815B is approved by ANSI.

This report has been reviewed and is approved.



Ada Validation Facility

Dale E. Lange

Technical Director

645 CCSG/SCSL

Wright-Patterson AFB OH 45433-5707

  
Ada Validation Organization

Director, Computer and Software Engineering Division

Institute for Defense Analyses

Alexandria VA 22311

  
Ada Joint Program Office

David R. Basel

Deputy Director

Defense Information Systems Agency,

Center for Information Management

## DECLARATION OF CONFORMANCE

Customer: TLD Systems, Ltd.

Ada Validation Facility: 645 C-CSG/SCSL  
Wright-Patterson AFB OH 45433-6503

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: TLD Comanche VAX/i960 Ada  
Compiler System, Version 4.1.1

Host Computer System: Digital Local Area Network VAX Cluster  
executing on (2) MicroVAX 3100 Model 90  
under VAX/VMS 5.5.

Target Computer System: Tronix JIAWG Execution Vehicle (i960MX)  
running TLD Real Time Executive (TLDrtx),  
(Domain Configuration), Version 4.1.1

### Customer's Declaration

I, the undersigned, representing TLD Systems, Ltd., declare that TLD Systems, Ltd. has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration executing in the default mode. The certificates shall be awarded in TLD Systems, Ltd.'s corporate name.

  
\_\_\_\_\_  
TLD Systems, Ltd.  
Terry L. Dunbar, President

Date: 10 February 1994

## TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	USE OF THIS VALIDATION SUMMARY REPORT . . . . .	1-1
1.2	REFERENCES. . . . .	1-2
1.3	ACVC TEST CLASSES . . . . .	1-2
1.4	DEFINITION OF TERMS . . . . .	1-3
CHAPTER 2	IMPLEMENTATION DEPENDENCIES	
2.1	WITHDRAWN TESTS . . . . .	2-1
2.2	INAPPLICABLE TESTS. . . . .	2-1
2.3	TEST MODIFICATIONS. . . . .	2-4
CHAPTER 3	PROCESSING INFORMATION	
3.1	TESTING ENVIRONMENT . . . . .	3-1
3.2	SUMMARY OF TEST RESULTS . . . . .	3-1
3.3	TEST EXECUTION. . . . .	3-2
APPENDIX A	MACRO PARAMETERS	
APPENDIX B	COMPILATION SYSTEM OPTIONS	
APPENDIX C	APPENDIX F OF THE Ada STANDARD	

## CHAPTER 1

### INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro92] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro92]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

#### 1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service  
5285 Port Royal Road  
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization  
Computer and Software Engineering Division  
Institute for Defense Analyses  
1801 North Beauregard Street  
Alexandria VA 22311-1772

## INTRODUCTION

### 1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language,  
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro92] Ada Compiler Validation Procedures, Version 3.1, Ada Joint  
Program Office, August 1992.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

### 1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK\_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK\_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK\_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values — for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

#### 1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.



## INTRODUCTION

Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro92].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

## CHAPTER 2

### IMPLEMENTATION DEPENDENCIES

#### 2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 22 November 1993.

B27005A	E28005C	B28006C	C32203A	C34006D	C35507K
C35507L	C35507N	C35507O	C35507P	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	C37310A	B41308B
C43004A	C45114A	C45346A	C45612A	C45612B	C45612C
C45651A	C46022A	B49008A	B49008B	A54B02A	C55B06A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
C83026A	B83026B	C83041A	B85001L	C86001F	C94021A
C97116A	C98003B	BA2011A	CB7001A	CB7001B	CB7004A
CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B	BD1B06A
AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A	CD4022A
CD4022D	CD4024B	CD4024C	CD4024D	CD4031A	CD4051D
CD5111A	CD7004C	ED7005D	CD7005E	AD7006A	CD7006E
AD7201A	AD7201E	CD7204B	AD7206A	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2117A	CE2117B
CE2119B	CE2205B	CE2405A	CE3111C	CE3116A	CE3118A
CE3411B	CE3412B	CE3607B	CE3607C	CE3607D	CE3812A
CE3814A	CE3902B				

#### 2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

## IMPLEMENTATION DEPENDENCIES

The following 201 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

C24113H..K (4 tests) have a line length greater than the maximum allowed line length of 120 for this implementation.

The following 20 tests check for the predefined type `LONG_INTEGER`; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than `INTEGER`, `LONG_INTEGER`, or `SHORT_INTEGER`; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`; for this implementation, there is no such type.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater; for this implementation, `MAX_MANTISSA` is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for `'SMALL` that are not powers of two or ten; this implementation does not support such values for `'SMALL`.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

D64005F..G (2) tests use 10 levels of recursive procedure calls nesting; this level of nesting for procedure calls exceeds the capacity of the compiler.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

CA3004E..F (2 tests) check that a program will execute when an optional body of one of its library packages is made obsolete; this implementation introduces additional dependences of the package declaration on its body as allowed by LRM 10.3(8), and thus the library unit is also made obsolete. (See Section 2.3.)

LA5007S..T (2 tests) check that a program cannot execute if a needed library procedure is made obsolete by the recompilation of a library unit named in that procedure's context clause; this implementation determines that the recompiled unit's specification did not change, and so it does not make the dependent procedure obsolete. (See Section 2.3.)

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files (See Section 2.3 regarding CE3413B):

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C

## IMPLEMENTATION DEPENDENCIES

CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A use an illegal file name in an attempt to create a file and expect NAME\_ERROR to be raised; this implementation does not support external files and so raises USE\_ERROR. (See section 2.3.)

## 2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 63 tests.

Note: CD2A81A is subject to two, distinct modifications as described below (the test name is marked with an asterisk).

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22005Z	B24009A	B25002A	B26005A	B44004D	B59001E
B73004B	B83033B	BA1020C	BA1020F	BA1101C	BA2001E
BA3006A	BA3013A				

C34009D and C34009J were graded passed by Evaluation Modification as directed by the AVO. These tests check that 'SIZE for a composite type is greater than or equal to the sum of its components' 'SIZE values; but this issue is addressed by AI-00825, which has not been considered; there is not an obvious interpretation. This implementation represents array components whose length depends on a discriminant with a default value by implicit pointers into the heap space; thus, the 'SIZE of such a record type might be less than the sum of its components 'SIZES, since the size of the heap space that is used by the varying-length array components is not counted as part of the 'SIZE of the record type. These tests were graded passed given that the Report.Result output was "FAILED" and the only Report.Failed output was "INCORRECT 'BASE'SIZE", from line 195 in C34009D and line 193 in C34009J.

C64104A, CB2006A, CB4002A, and CC1311B were graded passed by Processing Modification as directed by the AVO. These tests make various checks that CONSTRAINT\_ERROR is raised for certain operations when the resultant values lie outside of the range of the subtype. However, in many of the particular checks that these tests make, the exception-raising operation may be avoided as per LRM 11.6(7) by optimization that removes the operation if its only possible effect is to raise an exception (e.g., an assignment to a variable that is not later referenced). In the list below, beside the name of each affected test is given the line number of the check that is skipped (with a relevant associated operation's line number noted in parenthesis). These tests were processed both with and without optimization: the tests reported a passed result without optimization; with optimization, the checks cited below

## IMPLEMENTATION DEPENDENCIES

were skipped and a corresponding call to REPORT.FAILED was made.

C64104A	174 (copy back of parameter value)
CB2006A	36
CB4002A	85 (initialization @ 54)
CC1311B	55 (default parameter value @ 36)

C98001C was graded passed by Processing Modification as directed by the AVO. This test checks that a non-static argument to pragma Priority is not evaluated; it uses the pragma for the main program and within a task unit in the body of this program. This implementation evaluates the argument when the pragma appears in a task unit (at line 27) only; this behavior is in conformity to the draft revised Ada standard (a non-static argument will be illegal for a main program). (The AVO allows implementers to adopt Ada9X rules for Ada83 features so as to encourage the transition to the revised rules.) The test was processed with and without line 27 being commented out, and it reported "PASSED" and "FAILED" respectively.

CA3004E..F (2 tests) were graded inapplicable by Evaluation Modification as directed by the AVO. These tests check that a program will execute when an optional body of one of its library packages is made obsolete. This implementation, for optimization purposes, compiles all compilation units of a compilation into a single object module with a single set of control sections, collectively pooled constants, with improved addressing. As a consequence, the optional package body of these tests and its corresponding library unit have a mutual dependence, and thus the library unit is also made obsolete. This implementation-generated dependence is allowed by LRM 10.3(8).

LA5007S..T (2 tests) were graded inapplicable by Evaluation Modification as directed by the AVO. These tests check that a program cannot execute if a needed library procedure is made obsolete by the recompilation of a library unit named in that procedure's context clause. This implementation determines that the recompiled unit's specification did not change, and so it does not make the dependent procedure obsolete; the program executes, calling Report.Failed. The AVO ruled that this behavior is acceptable, in light of the intent for the revised Ada standard to permit such accommodating recompilation; further deliberation by the AVO and ARG will determine whether these (and many related) tests will be withdrawn.

The tests below were graded passed by Test Modification as directed by the AVO. These tests all use one of the generic support procedures, Length Check or Enum Check (in support files LENCHECK.ADA & ENUMCHEK.ADA), which use the generic procedure Unchecked Conversion. This implementation rejects instantiations of Unchecked Conversion with array types that have non-static index ranges. The AVO ruled that since this issue was not addressed by AI-00590, which addresses required support for Unchecked Conversion, and since AI-00590 is considered not binding under ACVC 1.11, the support procedures could be modified to remove the use of Unchecked Conversion. Lines 40..43, 50, and 56..58 in LENCHECK and lines 42, 43, and 58..63 in ENUMCHEK were commented out.

CD1009A	CD1009I	CD1009M	CD1009V	CD1009W	CD1C03A
CD1C04D	CD2A21A..C	CD2A22J	CD2A23A..B	CD2A24A	CD2A31A..C

## IMPLEMENTATION DEPENDENCIES

*CD2A81A	CD3014C	CD3014F	CD3015C	CD3015E..F	CD3015H
CD3015K	CD3022A	CD4061A			

\*CD2A81A, CD2A81B, CD2A81E, CD2A83A, CD2A83B, CD2A83C, and CD2A83E were graded passed by Test Modification as directed by the AVO. These tests check that operations of an access type are not affected if a 'SIZE clause is given for the type; but the standard customization of the ACVC allows only a single size for access types. This implementation uses a larger size for access types whose designated object is of type STRING. The tests were modified by incrementing the specified size \$ACC\_SIZE with '+ 64'.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as 'SMALL for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal 'SMALLs may be omitted.

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CE3413B was graded inapplicable by Evaluation Modification as directed by the AVO. This test includes the expression "COUNT'LAST > 150000", which raises CONSTRAINT\_ERROR on the implicit conversion of the integer literal to type COUNT since COUNT'LAST = 32,767; there is no handler for this exception, so test execution is terminated. The AVO ruled that this behavior was acceptable; the AVO ruled that the test be graded inapplicable because it checks certain file operations and this implementation does not support external files.

Many of the Class A and Class C (executable) test files were combined into single procedures ("bundles") by the AVF, according to information supplied by the customer and guidance from the AVO. This bundling was done in order to reduce the processing time—compiling, linking, and downloading to the target. For each test that was bundled, its context clauses for packages Report and (if present) SYSTEM were commented out, and the modified test was inserted into the declarative part of a block statement in the bundle. The general structure of each bundle was:

```
WITH REPORT, SYSTEM;  
PROCEDURE <BUNDLE_NAME> IS
```

— repeated for each test

```
DECLARE  
  <TEST FILE>    [a modified test is inserted here, ...]  
BEGIN  
  <TEST NAME>;  [... and invoked here]
```

```
EXCEPTION      —test is not expected to reach this exception handler  
  WHEN OTHERS => REPORT.FAILED("unhandled exception ");  
                REPORT.RESULT;
```

```
END;
```

## IMPLEMENTATION DEPENDENCIES

— [... repeated for each test in the bundle]

END <BUNDLE NAME>;

The 1259 tests that were processed in bundles are listed below; each bundle is delimited by '<' and '>'.

<A21001A	A22002A	A22006B	A26004A	A26007A	A27003A	A27004A
A29002A	A29002B	A29002C	A29002D	A29002E	A29002F	A29002G
A29002H	A29002I	A29002J	A29003A	A2A031A>	<A32203B	A32203C
A32203D	A33003A	A34017C	A35101B	A35402A	A35502Q	A35502R
A35710A	A35801A	A35801B	A35801F	A35902C	A38106D	A38106E
A38199A	A39005B	A39005C	A39005D	A39005E	A39005F>	<A39005G
A54B01A	A54B02A	A55B12A	A55B13A	A55B14A	A62006D	A71002A
A71004A	A72001A	A73001I	A73001J	A74105B	A74106A	A74106B
A74106C	A74205E	A74205F>	<A83009A	A83009B	A83041B	A83041C
A83041D	A83A02A	A83A02B	A83A06A	A83A08A	A83C01C	A83C01D
A83C01E	A83C01F	A83C01G	A83C01H	A83C01I	A83C01J	A85007D
A85013B	A87B59A>	<AB7006A	AC1015B	AC3106A	AC3206A	AC3207A>
<AD1A01A	AD1A01B	AD1D01E	AD7001B	AD7005A	AD7101A	AD7101C
AD7102A	AD7103A	AD7103C>	<AD7104A	AD7203B	AD7205B>	<C23001A
C23003A	C23006A	C24002A	C24002B	C24002C	C24003A	C24003B
C24003C	C24106A	C24113A	C24113B	C24113C	C24113D	C24113E>
<C24201A	C24202A	C24202B	C24202C	C24203A	C24203B	C24207A
C24211A	C25001A	C25001B	C25003A	C25004A	C26002B	C26006A>
<C26008A	C27001A	C2A001A	C2A001B	C2A001C	C2A002A	C2A006A
C2A008A	C2A009A	C2A021B>	<C32107A	C32107C	C32108A	C32108B
C32111A	C32111B>	<C32114A	C32115A	C32115B>	<C32117A	C34001A
C34001C	C34001D	C34001F	C34002A	C34002C	C34003A	C34003C>
<C34004A	C34004C	C34005A	C34005C>	<C34005D	C34005F	C34005G
C34005I>	<C34005J	C34005L	C34005M	C34005O>	<C34005P	C34005R
C34005S	C34005U	C34006A	C34006F	C34006G	C34006J>	<C34006L
C34007A	C34007D	C34007F	C34007G>	<C34007I	C34007J	C34007M
C34007P>	<C34007R	C34007S>	<C34009A	C34009F	C34009G	C34009L
C34011B	C34012A	C34014A	C34014C>	<C34014E	C34014G	C34014H
C34014J	C34014L	C34014N	C34014P	C34014R	C34014T>	<C34014U
C34014W	C34014Y	C34015B	C34016B	C34018A	C35003A	C35003B
C35003D	C35003F	C35102A	C35106A	C35404A>	<C35503A	C35503B
C35503C	C35503D	C35503E	C35503F	C35503G	C35503H	C35503K>
<C35503L	C35503O	C35503P	C35504A	C35504B	C35505A	C35505B
C35505C>	<C35505D	C35505E	C35505F	C35507A	C35507B>	<C35507C
C35507E	C35507G	C35507H	C35507I	C35507J>	<C35507K	C35507L>
<C35706A	C35706B	C35706C	C35706D	C35706E>	<C35707A	C35707B
C35707C	C35707D	C35707E	C35708A	C35708B	C35708C	C35708D
C35708E>	<C35711A	C35711B	C35712A	C35712B	C35712C	C35713A
C35713C>	<C35801D	C35802A	C35802B	C35802C	C35802D	C35802E>
<C35902A	C35902B	C35902D	C35904A	C35904B	C35A02A	C35A03A
C35A03B	C35A03C	C35A03D>	<C35A03N	C35A03O	C35A03P>	<C35A03Q
C35A04A	C35A04B	C35A04C>	<C35A04D	C35A04N>	<C35A04O	C35A04P>
<C35A04Q	C35A05A	C35A05D	C35A05N>	<C35A05Q	C35A06A	C35A06B>
<C35A06D	C35A06N	C35A06O>	<C35A06P	C35A06Q	C35A06R	C35A06S
C35A07A	C35A07B	C35A07C>	<C35A07D	C35A07N	C35A07O	C35A07P
C35A07Q	C35A08B	C36003A>	<C36004A	C36104A	C36104B	C36105B



# IMPLEMENTATION DEPENDENCIES

C36172A	C36172B	C36172C>	<C36174A	C36180A	C36202A	C36202B
C36202C	C36203A	C36204A	C36204B	C36204C>	<C36205A	C36205B
C36205C	C36205D	C36205E	C36205F	C36205G	C36205H>	<C36205I
C36205J	C36205K	C36301A	C36301B	C36302A	C36303A	C36304A
C36305A>	<C37002A	C37003A	C37003B	C37005A	C37006A	C37007A
C37008A	C37008B>	<C37008C	C37009A	C37010A	C37010B	C37012A
C37102B	C37103A	C37105A	C37107A	C37108B	C37206A	C37207A
C37208A	C37208B	C37209A	C37209B	C37210A>	<C37211A	C37211B
C37211C	C37211D	C37211E	C37213A	C37213B	C37213C	C37213D>
<C37213E	C37213F	C37213G	C37213H>	<C37213J	C37213K	C37213L
C37214A>	<C37215A	C37215B>	<C37215C	C37215D	C37215E	C37215F
C37215G	C37215H	C37216A	C37217A	C37217B	C37217C>	<C37304A
C37305A	C37306A	C37307A	C37309A	C37310A	C37312A	C37402A
C37403A>	<C37404A	C37404B	C37405A	C37409A	C37411A	C38002A
C38002B	C38004A	C38004B	C38005A	C38005B	C38005C	C38006A
C38102A	C38102B	C38102C	C38102D	C38102E	C38104A	C38107A
C38107B>	<C38108A	C38201A	C38202A	C39006A	C39006B	C39006D
C39006E	C39006G	C39007A	C39007B	C39008A	C39008B	C39008C>
<C41101D	C41103A	C41103B	C41104A	C41105A	C41106A	C41107A
C41108A	C41201D	C41203A	C41203B>	<C41204A	C41205A	C41206A
C41207A	C41301A	C41303A	C41303B	C41303C	C41303E	C41303F
C41303G	C41303I	C41303J	C41303K	C41303M	C41303N	C41303O
C41303Q	C41303R	C41303S	C41303U	C41303V	C41303W	C41304A>
<C41304B	C41306A	C41306B	C41306C	C41307A	C41307C	C41307D
C41308A	C41308C	C41308D	C41309A>	<C41320A	C41321A	C41322A
C41323A	C41324A	C41325A	C41326A	C41327A	C41328A>	<C41401A
C41402A	C41403A	C41404A	C42005A	C42006A	C42007A	C42007B>
<C42007C	C42007D	C42007E	C42007F	C42007G	C42007H	C42007I>
<C42007J	C42007K	C43003A	C43004B	C43103A	C43103B	C43104A>
<C43105A	C43105B	C43106A	C43107A	C43108A	C43204A	C43204C
C43204E	C43204F>	<C43204G	C43204H	C43204I	C43205A	C43205B
C43205C	C43205D	C43205E	C43205F	C43205G	C43205H	C43205I
C43205J	C43205K	C43206A	C43207A	C43207B	C43207C>	<C43207D
C43208A	C43208B	C43209A	C43210A	C43211A	C43212A	C43212C
C43213A>	<C43214A	C43214B	C43214C	C43214D	C43214E	C43214F
C43215A	C43215B	C43222A>	<C43224A	C44003A	C44003D	C44003E
C44003F	C44003G	C45101A	C45101B	C45101C	C45101E	C45101G
C45101H	C45101I	C45101K	C45104A	C45111A	C45111B	C45111C>
<C45111D	C45111E	C45112A	C45112B	C45113A>	<C45114B	C45122A
C45122B	C45122C	C45122D	C45123A	C45123B	C45123C>	<C45201A
C45201B	C45202A	C45202B	C45210A	C45211A	C45220A	C45220B
C45220C	C45220D	C45220E	C45220F	C45231A>	<C45232A	C45232B
C45241A	C45241B	C45241C	C45241D	C45241E>	<C45242A	C45242B
C45251A	C45252A	C45252B	C45253A	C45262A>	<C45272A	C45273A
C45274A	C45274B	C45274C	C45281A	C45282A	C45282B	C45291A
C45303A	C45304A>	<C45321A	C45321B	C45321C	C45321D	C45321E>
<C45323A	C45331A	C45331D	C45332A	C45342A	C45343A	C45344A
C45345A	C45345B	C45345C	C45345D>	<C45347A	C45347B	C45347C
C45347D	C45411A	C45411D	C45412A>	<C45413A	C45421A	C45421B
C45421C	C45421D	C45421E>	<C45423A	C45431A	C45502A	C45503A>
<C45504A	C45504D>	<C45505A	C45521A	C45521B	C45521C	C45521D
C45521E>	<C45523A	C45524A	C45524B	C45524C	C45524D	C45524E>
<C45532A	C45532B	C45532C	C45532D	C45532E	C45532F	C45532G
C45532H	C45532I	C45532J	C45532K	C45532L>	<C45534A	C45611A

## IMPLEMENTATION DEPENDENCIES

C45613A	C45614A	C45621A	C45621B	C45621C	C45621D	C45621E>
<C45622A	C45624A	C45624B	C45631A	C45632A	C45641A	C45641B
C45641C	C45641D	C45641E>	C45652A	C45662A	C45662B	C45672A
C46011A	C46012A	C46012B	C46012C>	<C46012D	C46012E>	<C46013A
C46014A	C46021A	C46023A	C46024A	C46031A	C46032A	C46033A>
<C46041A	C46042A	C46043A	C46043B>	<C46044A	C46044B	C46051A
C46051B	C46051C>	<C46052A	C46053A	C46054A	C47002A	C47002B
C47002C	C47002D	C47003A	C47004A	C47005A	C47006A	C47007A>
<C47008A	C47009A	C47009B	C48004A	C48004B	C48004C	C48004D
C48004E	C48004F	C48005A	C48005B	C48005C	C48006A	C48006B>
<C48007A	C48007B	C48007C	C48008A	C48008B	C48008C	C48008D
C48009A	C48009B	C48009C	C48009D	C48009E	C48009F	C48009G>
<C48009H	C48009I	C48009J	C48010A	C48011A	C48012A	C49020A
C49021A	C49022A	C49022B	C49022C	C49023A	C49024A	C49025A
C49026A>	<C4A005A	C4A005B	C4A006A	C4A007A	C4A010A	C4A010B
C4A010D	C4A011A	C4A012A	C4A012B	C4A013A	C4A013B	C4A014A>
<C51002A	C51004A	C52001A	C52001B	C52001C	C52005A	C52005B
C52005C	C52005D	C52005E	C52005F>	<C52007A	C52008A	C52008B
C52009A	C52009B	C52010A	C52011A	C52011B	C52012A	C52012B
C52013A>	<C52103B	C52103C	C52103F	C52103G	C52103H	C52103K
C52103L>	<C52103M	C52103P	C52103Q	C52103R	C52103S	C52103X
C52104A	C52104B	C52104C	C52104F>	<C52104G	C52104H	C52104K
C52104L	C52104M	C52104P	C52104Q	C52104R	C52104X	C52104Y>
<C53004B	C53005A	C53005B	C53006A	C53006B	C53007A	C53008A
C54A03A	C54A04A	C54A06A	C54A07A	C54A11A	C54A13A	C54A13B
C54A13C>	<C54A13D	C54A22A	C54A23A	C54A24A	C54A24B	C54A26A
C54A27A	C54A41A	C54A42A	C54A42B	C54A42C	C54A42D	C54A42E
C54A42F	C54A42G	C55B03A	C55B04A	C55B05A	C55B06A	C55B06B>
<C55B08A	C55B09A	C55B10A	C55B11A	C55B11B	C55B15A	C55B16A
C55C01A	C55C02A	C55C02B	C55C03A	C55C03B	C55D01A	C56002A
C57002A	C57003A	C57004A	C57004B	C57004C	C57005A>	<C58004A
C58004B	C58004C	C58004D	C58004F	C58004G	C58005A	C58005B
C58005H	C58006A	C58006B	C59001B	C59002A	C59002B	C59002C>
<C61008A	C61009A	C61010A	C62002A	C62003A	C62003B	C62004A
C62006A	C62009A	C63004A	C64002B>	<C64004G	C64005A	C64005B
C64005C	C64103A	C64103B	C64103C	C64103D	C64103E	C64103F>
<C64104A	C64104B	C64104C	C64104D	C64104E	C64104F	C64104G
C64104H	C64104I	C64104J	C64104K	C64104L	C64104M	C64104N
C64104O	C64105A	C64105B	C64105C	C64105D	C64105E	C64105F>
<C64106A	C64106B	C64106C	C64106D	C64107A	C64108A	C64109A
C64109B	C64109C	C64109D	C64109E>	<C64109F	C64109G	C64109H
C64109I	C64109J	C64109K	C64109L>	<C64201B	C64201C	C64202A
C65003A>	<C65003B	C65004A	C66002A	C66002C	C66002D	C66002E
C66002F	C66002G	C67002A	C67002B	C67002C	C67002D	C67002E>
<C67003A	C67003B	C67003C	C67003D	C67003E	C67005A	C67005B
C67005C	C67005D>	<C72001B	C72002A	C73002A	C73007A	C74004A
C74203A	C74206A	C74207B	C74208A	C74208B	C74209A	C74210A
C74211A	C74211B	C74302A	C74302B	C74305A	C74305B	C74306A
C74307A>	<C74401D	C74401E	C74401K	C74401Q	C74402A	C74402B
C74406A	C74407B	C74409B>	<C83007A	C83012D	C83022A	C83023A
C83024A	C83025A>	<C83027A	C83027C	C83028A	C83029A	C83030A>
<C83031A	C83031C	C83031E	C83032A	C83033A	C83051A	C83B02A
C83B02B	C83E02A	C83E02B	C83E03A	C83E04A	C83F01A	C83F03A
C84002A	C84005A	C84008A	C84009A	C85004B	C85005A	C85005B

# IMPLEMENTATION DEPENDENCIES

C85005C	C85005D>	<C85005E	C85005F	C85005G	C85006A>	<C85006F
C85006G>	<C87A05A	C87A05B	C87B02A	C87B02B	C87B03A	C87B04A
C87B04B	C87B04C	C87B05A	C87B06A	C87B07A	C87B07B>	<C87B07C
C87B07D	C87B07E	C87B08A	C87B09A	C87B09B	C87B09C	C87B10A
C87B11A	C87B11B	C87B13A	C87B14A	C87B14B	C87B14C	C87B14D>
<C87B15A	C87B16A	C87B17A	C87B18A	C87B18B	C87B19A	C87B23A
C87B24A>	<C87B24B	C87B26B	C87B27A	C87B28A	C87B29A	C87B30A
C87B31A	C87B32A>	<CB1001A	CB1002A	CB1003A	CB1004A	CB1005A
CB1010A	CB1010B	CB1010C	CB1010D>	<CB2004A	CB2005A	CB2006A
CB2007A	CB3003A	CB3003B>	<CB3004A	CB4001A	CB4002A	CB4003A
CB4004A	CB4005A	CB4006A	CB4007A	CB4008A	CB4009A	CB4013A
CB5002A	CB7003A	CB7005A>	<CC1004A	CC1005C	CC1010A>	<CC1010B
CC1018A	CC1104C	CC1107B	CC1111A	CC1204A	CC1207B	CC1220A
CC1221A	CC1221B	CC1221C	CC1221D>	<CC1222A	CC1224A	CC1225A>
<CC1304A	CC1304B	CC1305B	CC1307A	CC1307B	CC1308A	CC1310A>
<CC1311A	CC1311B	CC2002A	CC3004A	CC3007A	CC3011A	CC3011D
CC3012A	CC3015A	CC3106B>	<CC3120A	CC3120B	CC3121A	CC3123A
CC3123B	CC3125A	CC3125B	CC3125C	CC3125D>	<CC3126A	CC3127A
CC3128A	CC3203A	CC3207B	CC3208A	CC3208B>	<CC3208C	CC3220A
CC3221A	CC3222A	CC3223A	CC3224A	CC3225A>	<CC3230A	CC3231A
CC3232A	CC3233A	CC3234A	CC3235A	CC3236A	CC3240A	CC3305A
CC3305B	CC3305C	CC3305D	CC3406A	CC3406B	CC3406C	CC3406D
CC3407A	CC3407B	CC3407C	CC3407D	CC3407E	CC3407F>	<CC3408A
CC3408B	CC3408C	CC3408D	CC3504A	CC3504B	CC3504C	CC3504D
CC3504E	CC3504F>	<CC3504G	CC3504H	CC3504I	CC3504J	CC3504K>
<CC3601A	CC3601C>	<CC3603A	CC3606A	CC3606B	CC3607B>	

## CHAPTER 3

### PROCESSING INFORMATION

#### 3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical and sales information about this Ada implementation, contact:

Robert R. Risinger  
TLD Systems Ltd.  
3625 Del Amo Boulevard  
Torrance California 90503  
(310) 542-5433

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

#### 3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro92].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system — if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

## PROCESSING INFORMATION

a) Total Number of Applicable Tests	3534
b) Total Number of Withdrawn Tests	104
c) Processed Inapplicable Tests	67
d) Non-Processed I/O Tests	264
e) Non-Processed Floating-Point Precision Tests	201
f) Total Number of Inapplicable Tests	532 (c+d+e)
g) Total Number of Tests for ACVC 1.11	4170 (a+b+f)

### 3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the Serial Ports, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The following options were used for testing this implementation:

Compiler Option / Switch	Effect
NoPhase	Suppress displaying of phase times during compilation.
NoLog	To cause command line to be echoed on log file.
NoDebug	To suppress generation of debug symbols to speed compilation and linking.
List	To cause listing file to be generated.
Target=i960	Selects the TLD Intel i960 target architecture.

## PROCESSING INFORMATION

Linker Option / Switch	Effect
NoDebug	Suppresses generation of Debugger symbol files.
NoVersion	Suppresses announcement banners that contain timestamp and version information to facilitate file comparing.

All tests were executed with Code Straightening, Global Optimizations, and automatic Inlining options enabled. Where optimizations are detected by the optimizer that represent deletion of test code resulting from unreachable paths, deleteable assignments, or relational tautologies or contradictions, such optimizations are reflected by informational or warning diagnostics in the compilation listings.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

## APPENDIX A

### MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX\_IN\_LEN—also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$MAX_IN_LEN	120 — Value of V
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	'"' & (1..V/2 => 'A') & "'"
\$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & "'"
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"

# MACRO PARAMETERS

\$MAX\_STRING\_LITERAL    '' & (1..V-2 => 'A') & ''

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
\$ACC_SIZE	32
\$ALIGNMENT	4
\$COUNT_LAST	511
\$DEFAULT_MEM_SIZE	16#40000000#
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	I960
\$DELTA_DOC	2.0**(-31)
\$ENTRY_ADDRESS	15
\$ENTRY_ADDRESS1	17
\$ENTRY_ADDRESS2	19
\$FIELD_LAST	127
\$FILE_TERMINATOR	ASCII.FS
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
\$FORM_STRING	""
\$FORM_STRING2	CANNOT_RESTRICT_FILE_CAPACITY
\$GREATER_THAN_DURATION	90000.0
\$GREATER_THAN_DURATION BASE LAST	I31073.0
\$GREATER_THAN_FLOAT BASE LAST	3.41000E+38
\$GREATER_THAN_FLOAT_SAFE LARGE	2.I3000E+37



# MACRO PARAMETERS

\$GREATER\_THAN\_SHORT\_FLOAT\_SAFE\_LARGE  
 NO\_SUCH\_SHORT\_FLOAT\_TYPE  
 \$HIGH\_PRIORITY 20  
 \$ILLEGAL\_EXTERNAL\_FILE\_NAME1  
 "BADCHAR@!"  
 \$ILLEGAL\_EXTERNAL\_FILE\_NAME2  
 "THISFILENAMEWOULDBEPERFECTLYLEGAL" &  
 "IFITWERENOTSOLONG.SOTHERE"  
 \$INAPPROPRIATE\_LINE\_LENGTH  
 -1  
 \$INAPPROPRIATE\_PAGE\_LENGTH  
 -1  
 \$INCLUDE\_PRAGMA1 PRAGMA INCLUDE ("A28006D1.TST")  
 \$INCLUDE\_PRAGMA2 PRAGMA INCLUDE ("B28006F1.TST")  
 \$INTEGER\_FIRST -2147483648  
 \$INTEGER\_LAST 2147483647  
 \$INTEGER\_LAST\_PLUS\_1 2147483648  
 \$INTERFACE\_LANGUAGE ASSEMBLY  
 \$LESS\_THAN\_DURATION -90000.0  
 \$LESS\_THAN\_DURATION\_BASE\_FIRST  
 -131073.0  
 \$LINE\_TERMINATOR ACSII.CR  
 \$LOW\_PRIORITY 1  
 \$MACHINE\_CODE\_STATEMENT  
 CTRL'(B, 1, True)  
 \$MACHINE\_CODE\_TYPE CTRL  
 \$MANTISSA\_DOC 31  
 \$MAX\_DIGITS 15  
 \$MAX\_INT 2147483647  
 \$MAX\_INT\_PLUS\_1 2147483648  
 \$MIN\_INT -2\_147\_483\_648

# MACRO PARAMETERS

\$NAME	NO_SUCH_INTEGER_TYPE
\$NAME_LIST	Pmachine, ns16000, vax, af1750 z8002, z8001, gould, pdp11, m68000, pe3200, caps, amdaul, i8086, i80286, i80386, z80000, ns32000, ibms1, m68020, nebula, name_x, hp, bbl, hawk, rl666, i960
\$NAME_SPECIFICATION1	Not supported
\$NAME_SPECIFICATION2	Not supported
\$NAME_SPECIFICATION3	Not supported
\$NEG_BASED_INT	16#FFFFFFFE#
\$NEW_MEM_SIZE	16#10000000#
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	i960
\$PAGE_TERMINATOR	ACSII.CR & ASCII.FF
\$RECORD_DEFINITION	Withdrawn
\$RECORD_NAME	Withdrawn
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	2000
\$TICK	0.000001
\$VARIABLE_ADDRESS	SYSTEM."-(16#7FFFFFFF4#)
\$VARIABLE_ADDRESS1	SYSTEM."-(16#7FFFFFFEC#)
\$VARIABLE_ADDRESS2	SYSTEM."-(16#7FFFFFFE8#)
\$YOUR_PRAGMA	Withdrawn

## APPENDIX B

### COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

---

### 3.4 COMPILER OPTION SWITCHES

Compiler option switches provide control over various processing and output features of the compiler. These features include several varieties of listing output, the level and kinds of optimizations desired, the choice of target computer, and the operation of the compiler in a syntax checking mode only.

Keywords are used for selecting various compiler options. The complement keyword, if it exists, is used to disable a compiler option and is formed by prefixing the switch keyword with "NO".

Switch names may be truncated to the least number of characters required to uniquely identify the switch. For example, the switch "CROSSREF" (explained in the list below) may be uniquely identified by the abbreviation "CR" or any longer abbreviation. In the list of switches on the following pages, the abbreviations are in bold and the optional extra characters are not bolded.

If an option is not specified by the user, a default setting is assumed. All specified compiler options apply to a single invocation of the compiler.

The default setting of a switch and its meaning are defined in the table below. The meaning of the complement form of a switch is normally the negation of the switch. For some switches, the complement meaning is not obvious; these complement switch keywords are listed separately.

In the description of the switches, the target dependant name target is used. The value of this symbol is determined by the value of the TARGET switch.

Compiler-generated file specifications generally conform to host conventions. Thus, any generated filename is the source filename appended with the default file type. The output file name can be completely or partially specified.

**SWITCH NAME**

**MEANING**

---

**ADDRESS\_SPACE=name** (name, subsystem\_name)  
**NOADDRESS\_SPACE** -- default

This switch allows users to specify the association of a compilation unit with a logical address space. This capability will support the definition of i960 Extended Architecture "Domains" and domain calls.

The name parameter is the name of the address space and subsystem-name is the name of the subsystem to which the address space belongs. If subsystem-name is not supplied, then the address space does not belong to a subsystem. This switch may appear in any compilation, and applies to all the compilation units in the compilation.

**NOTE:** An alternate method of associating compilation unit(s) with a logical address space is to use the pragma Address\_Space in the compilation unit(s) and compile without using this switch. The pragma Address\_Space\_Entry is used to indicate which subprograms represent entities into the logical space (defined by this switch or pragma Address\_Space). Refer Section 5.2.F of this document under Implementation-Dependent Pragmas, for further information.

This capability does not yet allow users to indicate objects that are to be implemented and referenced as independent objects.

TLDlnk will verify that all compilation units in the link have an address space attribute of the same value, or have no address space attribute and will create either a domain (if an address attribute is specified) or a program (if no address space attribute is specified).

**AIId=adb-filename**  
**NOAIId** -- default

This switch causes information collected during compilation to be saved in a specified data base file or a default file named I960.ADB in the compilation directory. This information includes the compilation units, the contained scopes, the local declarations of objects and types and their descriptions, external references, callers, calls, program design language (PDL) which is extracted from stylized Ada comments embedded in the source code, and any other information extracted from similar stylized Ada comments. The TLD Ada Info Display (TLDaid) permits the user to browse this data base and to extract selected data base information to support the understanding of a program or to produce documentation describing the program.

CALL TREE  
NOCALL\_TREE -- default

This switch is used in conjunction with ELABORATOR and LIST to cause all .CTI files (corresponding to the complete set of object files being linked for this program) to be read in and a closure of all calls in the program to be computed. The results of this analysis is formatted into a subprogram call tree report and output in the listing file. This switch has no effect without the ELABORATOR and LIST switches.

NOTE: The call tree is incomplete if any required compilation unit's .CTI files are missing.

CHECKS -- default  
CHECKS{=(check\_identifier{,...})}  
NOCHECKS{=(check\_identifier{,...})}

When the CHECKS switch is used, zero or more check\_identifiers are specified and the run time checks are enabled. The status of run time checks associated with unmentioned check\_identifiers is unchanged.

Without any check\_identifiers, the NOCHECKS switch omits all run time checks. If one or more check\_identifiers are specified, the specified run time checks are omitted. The status of run time checks associated with unmentioned check\_identifiers is unchanged.

Checks can be eliminated selectively or completely by source statement pragma Suppress. Pragma Suppress overrides the CHECKS switch.

Check\_identifiers are listed below and are described in the LRM, Section 5.2.B.

ALL\_CHECKS -- default (consists of all the checks below)

ACCESS_CHECK	DISCRIMINANT_CHECK	DIVISION_CHECK
ELABORATION_CHECK	INDEX_CHECK	LENGTH_CHECK
OVERFLOW_CHECK	RANGE_CHECK	STORAGE_CHECK

**COMMENT-character-specification(...)**

This switch allows the user to override a set of default meta characters used to mark comments which have special meanings to the Compiler. (In the source code, these meta characters must immediately follow the Ada comment designator "...".) There are 13 meta characters defined as positional entries in the string of characters specified for this switch. To define one or more entries, all entries up to and including the last entry to be defined must be specified. Each of these characters may be represented either by the character itself, or by a dollar sign "\$" followed by the character's decimal ascii value. (The latter form is useful for specifying characters which would otherwise be significant to the command line parser.) To specify a dollar sign character, use the form "\$36". Remaining character positions are left unchanged. Capabilities for character positions in the string may be disabled by specifying either blank ("32") or null ("0"). Please refer to the ascii character set table in Appendix B for the decimal value of ascii characters. The definition of each entry and its current default value is as follows:

**TLDada Conditional Compilation Comment Characters**

**Position Description**

- 1      **Configuration Equals**    (default: "=")  
          This entry defines the character used to mark conditional source lines which will be included in the compilation only if its configuration-identifier is specified with the CONFIGURATION switch.
  
- 2      **Configuration Not Equal**    (default: "#")  
          This entry defines the character used to mark conditional source lines which will be included in the compilation only if its configuration-identifier is not specified with the CONFIGURATION switch. This same character is used to begin an "else" clause within a group of conditional compilation lines. The lines between this character and the end of the group will be included in the compilation only if the configuration-identifier for the group is not specified with the CONFIGURATION switch.
  
- 3      **Begin Configuration**    (default: "(")  
          This entry defines the character used to mark the beginning of a group of conditional compilation source lines.

- 4      **End Configuration**    (default: "}")  
         This entry defines the character used to mark the  
         end of a group of conditional compilation source  
         lines.

See the **CONFIGURATION** command line option for more information on  
conditional compilation.

**Source Reformatting Comment Characters**

**Position    Description**

- 5      **Continuation Line**    (default: "&")  
         This entry defines the character used to mark a  
         comment continued from the previous line and for  
         which word-wrapping is performed during source code  
         reformatting.
- 6      **Reserved for future use.**
- 7      **Reserved for future use.**

**TLDaid Comment Characters**

**Position    Description**

- 8      **Begin Topic**    (default: "[")  
         This entry defines the character used to mark the  
         beginning of text associated with a topic name.
- 9      **End Topic**    (default: "]")  
         This entry defines the character used to mark the  
         end of text associated with a topic name.
- 10     **Define Topic**    (default: "@")  
         This entry defines the character used to mark the  
         definition of a comment meta character for a  
         particular user-defined topic name. This character  
         may subsequently be used as a shorthand for the  
         above method, eliminating the need to specify the  
         topic name at each occurrence.
- 11     **Description**    (default: ":")  
         This entry defines the character used to mark a  
         comment as a description associated with the  
         previous declaration.



- 12 PDL (default: "|")  
This entry defines the character used to mark a comment as Program Design Language (PDL).
- 13 Command (default: "\$")  
This entry defines the character used to mark a comment as a command to control data collection. This entry provides a mechanism for users to maintain compatibility between the new implementation and previously commented source. It also may be used to provide a degree of compatibility with tools similar to TLDaid.

See the Reference Document for the TLD Utilities for more information on TLDaid.

CONFIGURATION={()configuration-identifier{...}{}}

where the parenthesis ( ) are required only when more than one configuration-identifier is specified.

This switch provides a conditional compilation (configuration) capability by determining the specially commented source lines that are to be included in the compilation. Source line(s) can be associated with a configuration-identifier which if supplied with this switch, causes them to be included. Also, alternative source line(s) can be specially marked to be included if the configuration-identifier is not supplied.

#### Format

Mark Source Line(s) Individually:

--configuration-identifier      conditional-source-line

or:

--#configuration-identifier      conditional-source-line

The above format is repeated for each source line to be marked as a conditional source line.

Source line(s) beginning with "--" are included in the compilation if the configuration-identifier is specified with the command line CONFIGURATION switch. Source line(s) beginning with "--#" are included in the compilation if the configuration-identifier is not specified with the command line CONFIGURATION switch (CONFIGURATION is not used or is used without that configuration-identifier).

**Mark a Group of Source Lines:**

```
--{configuration-identifier
conditional-source-line-1      }
.                               } Compiled if
.                               } configuration-identifier
.                               } is specified with this
conditional-source-line-n      } switch.
--#
alt-conditional-source-line-1   }
.                               } Compiled if
.                               } configuration-identifier
.                               } is not specified with this
alt-conditional-source-line-n   } switch.
--}configuration-identifier
```

Source line(s) between lines beginning with "--{" and "--#" are included in the compilation if the configuration-identifier is specified with the command line CONFIGURATION switch. Source line(s) between lines beginning with "--#" and "--}" are included in the compilation if the configuration-identifier is not specified with the command line CONFIGURATION switch (CONFIGURATION is not used or is used without that configuration-identifier).

**Notes on Syntax**

Comments are examined for configuration switches only if they occupy a line by themselves (i.e., the "--" starts at the first non-blank character of the line).

The special comment characters "--=", "--#", "--{", and "--}" must be entered as shown with no spaces between them.

The characters "=", "#", "{", and "}" are the default meta characters for configuration switches, but they can be modified. See the COMMENT command line option for more information.

The configuration-identifier must immediately follow the special comment characters; no space is allowed between them.

The configuration-identifier on the closing brace "--}" is optional, but if specified must match the identifier on the corresponding opening brace "--{".

The "--#" has one or the other of two distinct meanings: 1) if followed by a configuration-identifier, it means "compile the balance of this line conditionally" and 2) if no identifier follows, it means "toggle the sense of the innermost configuration brace".

Any additional text on the same line as the "--{configuration-identifier" and/or the "--} configuration-identifier" will be considered a comment and will not be compiled as Ada source, regardless of the configuration settings.

#### Naming Constraint

By default, a /CONFIG-I960 setting is created for the target computer and model (by the /TARGET and the /MODEL Compiler switches). Therefore, I960 is not a valid configuration-identifier for conditional compilation. If used, conditional source with that name will always be included in the compilation whether or not this switch is specified (since that name is already specified for the target and model, by default).

#### Nesting

The compiler treats nested conditional source in a manner similar to nested "if" statements. It checks the configuration-identifier to determine if it has been specified with the CONFIGURATION switch (similar to the checking performed to determine whether an "if" statement is to be performed). If so, it selects the source marked with that configuration-identifier (just as an "if" statement is performed for a "True" "if" condition). If not and alternate conditional source exists, it selects the alternate source for that configuration-identifier marked with "--#" (just as an "else" statement is performed for a "False" "if" condition). It continues this checking for every nested configuration-identifier it encounters.

For example:

```
--{A
  conditional-source-line-A1      }
      .                          } Compiled if A is
      .                          } specified with
      .                          } this switch.
  conditional-source-line-An      }
--#
  alt-conditional-source-line-A1   }
      .                          } Compiled if A is not
      .                          } specified with
      .                          } this switch.
  alt-conditional-source-line-An   }
--{B
  conditional-source-line-B1      }
      .                          } Compiled if A is not
      .                          } and B is specified
      .                          } with this switch.
  conditional-source-line-Bn      }
--#
  alt-conditional-source-line-B1   }
      .                          } Compiled if A and B are
      .                          } not specified with
      .                          } this switch.
  alt-conditional-source-line-Bn   }
--{C
  conditional-source-line-C1      }
      .                          } Compiled if A and B are
      .                          } not and C is specified
      .                          } with this switch.
  conditional-source-line-Cn      }
--#
  alt-conditional-source-line-C1   }
      .                          } Compiled if A, B, and C
      .                          } are not specified with
      .                          } this switch.
  alt-conditional-source-line-Cn   }
--}C
--}B
--}A
```

Configuration switches are examined and must be properly nested regardless of whether or not the configuration-identifiers are specified.

The following example format is invalid, since "B" is not completely nested within "A":

```
--{A
--{B
--}A
--}B
```

At the close of "A", the nested "B" will be forced closed with the warning message: "Missing configuration comment: --}B". By the time "--}B" is reached, "B" will have already been closed, so the following warning will be issued: "Unmatched configuration comment: --}B".

**CROSSREF**  
**NOCROSSREF** -- default

This switch generates a cross reference listing that contains names referenced in the source code. The cross reference listing is included in the listing file; therefore, the LIST switch must be selected or CROSSREF has no effect.

**CTI**  
**NOCTI** -- default

This switch generates a CASE tools interface file. The default filename is derived from the object filename, with a .CTI extension. The .CTI file is required to support the CALL\_TREE, FULL\_CALL\_TREE, and INVERTED\_CALL\_TREE switches.

**DEBUG** -- default  
**NODEBUG**

This switch selects the production of symbolic debug tables in the relocatable object file.

Alternate abbreviation: DBG, NODEBG

**DIAGNOSTICS**  
**NODIAGNOSTICS** -- default

This switch produces a diagnostic message file compatible with Digital's Language Sensitive Editor and XincTech Editor. See Digital's documentation for the Language Sensitive Editor for a detailed explanation of the file produced by this switch.

**ELABORATOR**  
**NOELABORATOR** -- default

This switch generates a setup program (in unit-name\$ELAB.OBJ (and a listing file in unit-name\$ELAB.LIS if the LIST switch was specified)) that elaborates all compilation units on which the specified library unit procedure (main program) depends and then calls the procedure (main program). When the ELABORATOR switch is used, The unit name of a previously compiled procedure must be specified instead of a source file. It is not necessary to distinguish a main program from a library unit when it is compiled.

**FULL\_CALL\_TREE**  
**NOFULL\_CALL\_TREE** -- default

When the FULL\_CALL\_TREE switch is used, the compiler listing includes all calls including all nested calls in every call. The NOFULL\_CALL\_TREE switch shows all nested calls in the first instance only and all subsequent calls are referred to the first instance. This switch has no effect without the ELABORATOR and LIST switches.

**INDENTATION=n**  
**INDENTATION=3** -- default

This switch controls the indentation width in a reformatted source listing (see the REFORMAT switch description). This switch assigns a value to the number of columns used in indentation; the value n can range from 1 to 8.

**INFO** -- default  
**NOINFO**

The INFO switch produces all diagnostic messages including information-level diagnostic messages. The NOINFO switch suppresses the production of information-level diagnostic messages only.

**INSTANTIATE=option**  
**NOINSTANTIATE** -- default

This switch is used to establish a default mode of instantiation for all generic instantiations within the compilation.

The option parameter instructs the Compiler to instantiate generics in the manner specified, as described below:

single\_body - a single body is used for all instantiations

macro - each instantiation produces a different body

Please refer to Section 3.12 "Generics" for more information on the advantages and disadvantages in using single\_body generics versus macro generics.

Nested instantiations and nested generics are supported and generics defined in library units are permitted.

It is not possible to perform a macro instantiation for a generic whose body has not yet been compiled.

NOTE: An alternate method of controlling macro instantiation of a generic is by using pragma Instantiate in the source code and performing compilation without this switch. The pragma controls instantiation of a particular generic. Refer to Section 5.2.F of this document under Implementation-Dependent Pragmas, for further information.

In the event of a conflict between the pragma and this switch, the switch takes precedence.

**INTSL**  
**NOINTSL** -- default

This switch intersperses lines of source code with the assembly code generated in the macro listing. This switch is valid only if the LIST and MACRO switches are selected. It may be helpful in correlating Ada source to generated code, but it increases the size of the listing file.

**INVERTED\_CALL\_TREE**  
**NOINVERTED\_CALL\_TREE** -- default

This switch determines which calls led to the present one. A reversed order call tree is generated. This switch has no effect without the ELABORATOR and LIST switches.

**LIST**{-listing-file-spec}  
**NOLIST** -- default in interactive mode  
**LIST** -- default for background processes

This switch generates a listing file. The default filename is derived from the source filename, with a .LIS extension. The listing-file-spec can be optionally specified.

**LOG**  
**NOLOG** -- default

This switch causes the compiler to write in the compilation log, command line options and the file specification of the Ada source file being compiled which is written to to SYS\$OUTPUT (the operating system's standard output). This switch is useful in examining batch output logs because it allows the user to easily determine which files are being compiled.

**MACRO**  
**NOMACRO** -- default

This switch produces an assembly like object code listing appended to the source listing file. The LIST switch must be enabled or this switch has no effect.

**MAIN\_ELAB**  
**NOMAIN\_ELAB** -- default

This switch makes the compiler treat the compilation unit being compiled as a user-defined elaboration or setup program which is used instead of that normally produced by the ELABORATOR switch. The source file must be specified instead of a unit name of a previously compiled procedure. Usually, the source file is modified by the user, starting from the version produced by the WRITE\_ELAB switch.

**MAXERRORS=n**  
**MAXERRORS=500** -- default

This switch assigns a value limit to the number of errors forcing job termination. Once this value is exceeded, the compilation is terminated. Information-level diagnostic messages are not included in the count of errors forcing termination. The specified value's range is from 0 to 500.

**MODEL=model-name**

If this switch is not specified, TLDada provides compilation capabilities that are common to all models of the target.

If this switch is specified, where model-name is one of the models below, TLDada provides compilation capabilities that are valid for the specified model. The compilation that is performed for a particular model may be valid for another model of the target if it supports the same machine-specific code (machine instructions, domains, etc.).



The following are valid models:

KA  
SA  
CA  
KB  
SB  
MC  
MM  
XA  
MX

**NEW\_LIBRARY**  
**NONEW\_LIBRARY** -- default

The **NEW\_LIBRARY** switch creates an I960 subdirectory in your current working directory and an I960.LIB library in that subdirectory, replacing the contents of the prior subdirectory and library, if they existed.

The **NONEW\_LIBRARY** switch checks if an I960 subdirectory exists in your current working directory and if it does not already exist, it will create the I960 subdirectory and an I960.LIB library in that subdirectory.

**NOTE:** This switch along with the **PARENT\_LIBRARY** switch replaces the **MAKE\_LIB** switch.

**OBJECT**(=object-file-spec)  
**OBJECT** -- default  
**NOOBJECT**

This switch produces a relocatable object file in the I960 subdirectory in the current compilation directory. The default filename is derived from the source filename, with a ".OBJ" extension.

**OPT** -- default  
**OPT**(= (parameter{,...}))  
**NOOPT**  
**NOOPT**(= (parameter{,...}))

This switch enables the specified global optimization of the compiled code. The negation of this switch disables the specified global optimization of the compiled code.

When the OPT switch is entered, without any parameters, all optimizations listed below are turned on. This restores the parameters to their defaults. When it is entered with parameters, only the specified parameters are turned on.

When the NOOPT switch is entered, without any parameters, all optimizations listed below are turned off. When it is entered with parameters, only the specified parameters are turned off.

Default optimizations such as COMMON\_SUBEXPRESSION, CONSTANT\_ARITHMETIC, DEAD\_CODE, and VALUE\_FOLDING, etc. should not be changed for normal use. Users may wish to change these optimizations for configuration or testing purposes, however, TLD Systems recommends that they not be changed. These default optimizations should be changed only when there is an abnormal situation with data or the program or a bad, TLD- or user-created algorithm. For example, if the program has an unused procedure the default optimization parameter DEAD\_SUBPROGRAM default will delete it for production improvement, however, the user may not want the unused procedure deleted for Debugger purposes. If users are finding a need to change these optimizations, please notify TLD Systems so that we can resolve the problem more efficiently.

The following parameters may be used with the /OPT and /NOOPT switches:

### **CODE\_Movement**

This parameter moves code to improve execution time. (For example, moves invariant code out of a loop). This parameter is turned on by default.

### **CODE\_STRAIGHTENING**

This parameter ensures that program flow is well formed by performing rearrangement of segments of code. This parameter is turned on by default.

### **COMMON\_SUBEXPRESSION**

Expressions with the same operands are not computed a second time. (For example, if an expression uses "A + B" and another expressions uses "A + B", the Compiler does not compute the second expression, since it knows it has already computed the value). This parameter is turned on by default. **WARNING:** Turning this switch off may cause unexpected results.

### **CONSTANT\_ARITHMETIC**

This parameter performs constant arithmetic. This parameter is turned on by default. **WARNING:** Turning this switch off may cause unexpected results.

### **DEAD\_Code**

This parameter removes code that cannot be reached such as unlabeled code following an unconditional branch. This parameter is turned on by default. **WARNING:** Turning this switch off may cause unexpected results.

### **DEAD\_SubPROGRAM**

This parameter removes subprograms that are not referenced. This parameter is turned on by default.

### **DEAD\_VARIABLE**

This parameter removes local temporary variables that are not used during execution. This parameter is turned on by default.

### **DELASSIGN**

This parameter optimises code by deleting redundant assignments. It only performs deletions allowed by the semantics of Ada. This parameter is turned on by default.

### **INLINE**

By default, the compiler automatically inlines subprograms that are not visible in a package spec and if the estimated code size is smaller than the actual call, it will inline it. This parameter is turned on by default.

### **LITERAL\_POOL**

This parameter overrides the Compiler's optimization separation of compile time constants into a separate memory pool. This parameter enables the user to exercise complete control over data allocation. This parameter is turned on by default.

### **LOOP\_UNROLLING**

This parameter applies to register memory only. It causes an expression computed at the end of a loop to be remembered at the top of the next iteration. This parameter is turned on by default.

### **PEEKHOLE**

This parameter performs optimization in very limited contexts. This parameter is turned on by default.

### **REGISTER\_DEDICATION**

This parameter allows dedication of a register to an object or expression value. This parameter is turned on by default.

### **SCHEDULER**

This parameter is used to activate the reorganizer phase of the Compiler. Instruction Scheduling, as performed by the Reorganizer, is a phase between the Code Generator and the Object Formatter phases. The Reorganizer reads the Code File, reorders the code, and outputs the Code File. This parameter is turned off by default.

The purpose of the Reorganizer is to perform optimization on the code generated by the Code Generator in order to minimize the amount of time that the hardware has to wait for data, generated by earlier instructions, to become ready for use.

**NOTE:** If you choose to use this switch, TLD recommends that the System Administrator set the user's page file quota to at least 60,000.

### **SINGLE\_MODULE**

This parameter creates one object module per compilation unit rather than one for each top-level subprogram. If this parameter is not used, and the compilation unit spec and body are in separate files, the extension "\_b" is added to the package name in the object file name of the package body (i.e., package-name\_b.obj) to differentiate between the package body and spec. The user may locate csects from only the body or spec by specifying the unique object filename (package-name\_b for the body or package-name for the spec) followed by the control section name. This parameter is turned on by default.

### **STRENGTH\_REDUCTION**

This parameter selects operators that execute faster. This parameter is turned on by default.

### **VALUE\_FOLDING**

Substitutions of operands known to have the same value are performed before expression analysis optimization. (For example, if B and C have the same value, the expression "A + C" is used and "A + B" will be recognized as common and the Compiler will not compute the second expression, since it knows it has the same value as the first). This parameter is turned on by default. **WARNING:** Turning this switch off may cause unexpected results.

**PAGE-lines-per-page**  
**PAGE=60 -- default**

This switch assigns a value to the number of lines per page for listing. The value can range from 10 to 99.

**PARENT\_LIBRARY-parent-library-spec**  
**NOPARENT\_LIBRARY -- default**

The **PARENT\_LIBRARY** switch uses the specified library as the parent library for the library to be created. 1750A must be included at the end of the parent-library-spec. This switch may only be used with the **NEW\_LIBRARY** switch.

If the **NOPARENT\_LIBRARY** switch is used, the library created by the **NEW\_LIBRARY** switch will have no parent library.

**NOTE:** This switch along with the **NEW\_LIBRARY** switch replaces the **MAKE\_LIB** switch.

**PARMs**  
**NOPARMs -- default**

This **PARAMETER** switch causes all option switches governing the compilation, including the defaulted option switches, to be included in the listing file. The **LIST** option switch must also be selected or this switch has no effect. User specified switches are preceded in the listing file by a leading asterisk (\*). This switch adds approximately one page to the listing file.

**PHASE -- default**  
**NOPHASE**

This switch suppresses the display of phase names during compilation. This switch is useful in batch jobs because it reduces the verbosity of the batch log file.

REF ID CASE=option  
NOREF ID CASE=option -- default

This is a reformatting option, under the control of the REFORMAT switch. This switch determines how variable names appear in the compiler listing. The options for this switch are:

ALL_LOWER	-- All variable names are in lower case.	
ALL_UPPER	-- All variable names are in upper case.	
INITIAL_CAPS	-- All variable names have initial caps.	-- default

REF KEY CASE=option  
NOREF KEY CASE=option -- default

This is a reformatting option, under the control of the REFORMAT switch. This switch determines how Ada key words appear in the compiler listing. The options for this switch are:

ALL_LOWER	-- All Ada key words are in lower case.	-- default
ALL_UPPER	-- All Ada key words are in upper case.	
INITIAL_CAPS	-- All Ada key words have initial caps.	

REFORMAT{-reformat-file-spec}  
NOREFORMAT -- default

This switch causes the compiler to reformat the source listing in the listing file (if no reformat-file-spec was provided) or generate a reformatted source file, if a reformat-file-spec is present. The default file extension of the reformatted source file is ".RFM". Reformatting consists of uniform indentation and retains numeric literals in their original source form. This switch performs the reformatting as specified by the REF\_ID\_CASE, REF\_KEY\_CASE, and INDENTATION switches.

SOURCE -- default  
NOSOURCE

This switch causes the input source program to be included in the listing file. Unless they are suppressed, diagnostic messages are always included in the listing file.

SYNTAX\_ONLY  
NOSYNTAX\_ONLY -- default

This switch performs syntax and semantic checking on the source program. No object file is produced and the MACRO switch is ignored. The Ada Program Library is not updated.

**TARGET=i960 -- default**

This switch selects the target computer for which code is to be generated for this compilation. "i960" selects i960 architecture operation.

**WARNINGS -- default**  
**NOWARNINGS**

The WARNINGS switch outputs warning and higher level diagnostic messages.

The NOWARNINGS switch suppresses the output of both warning-level and information-level diagnostic messages.

**WIDTH=characters-per-line**  
**WIDTH=110 -- default**

This switch sets the number of characters per line (80 to 132) in the listing file.

**WORD\_STORE**  
**NOWORD\_STORE -- default**

The WORD\_STORE switch simulates byte and half-word stores by using full word instructions. This will allow only full word stores to be performed. The NOWORD\_STORE switch will allow byte and half-word stores to be performed.

**WRITE\_ELAB**  
**NOWRITE\_ELAB -- default**

The WRITE\_ELAB switch generates an Ada source file which represents the main elaboration "setup" program created by the compiler. The unit name of a previously compiled procedure must be specified instead of a source file. The WRITE\_ELAB switch may not be used at the same time as the ELABORATOR switch.

**XTRA**  
**NOXTRA -- default**

This switch is used to access features under development or features not defined in the LRM. See the description of this switch in Section 3.15.

## COMPILATION SYSTEM OPTIONS

### LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.



## 4 DIRECTIVE LANGUAGE

---

On any host, the command line calling TLDlnk may optionally specify a linker directive file to control the linking operation. The directive file format and individual directives are described in the following pages.

---

### 4.1 DIRECTIVE FILE

Each line of the directive file contains up to 132 characters. Tabs are treated as blanks. Blanks are used to separate words when no other punctuation separates them; the actual number of blanks is insignificant. Characters that follow two successive minuses (--) are ignored.

A directive ordinarily consists of one line of input, however, lines may be continued using a tilde (-). Only one directive is allowed per input line. A continuation character can be used to continue directive values, however, a value cannot be split between two lines (if the value does not fit on the current line, the continuation character must be used after the previous value and the value must be placed on the following line). Either upper or lower case may be used (they are equivalent) except for file names on hosts with case-sensitive file names.

A directive file may include another directive file. The format of this directive is:

```
INCLUDE filespec(.lnk)
```

where the file extension .lnk may be optionally specified if the file is a directive file, otherwise, if the file is an object module file, its extension must be supplied (i.e., .obj or .olb must be supplied).

An included file may itself include another directive file, that is, in the example above, filespec.lnk may contain yet another directive file. The level of nested directive files allowed depends on the number of files that the operating system permits to be opened simultaneously.

Conditional linking may be performed in the directive file. The format of this conditional linking is:

```
if logical-expression then
...
{else | elsif logical-expression then}
...
endif
```

If the logical-expression returns a true value, the statements following the if or the elsif clause will be processed, otherwise, the ones following the else clause will be processed.

---

## **4.2 DIRECTIVES**

TLDLink directives are described in this section, in alphabetical order.

The following words, in lower case italics, are used in the descriptions:

*csect-name*

This is the name of the control section in the program being linked.

*file*

This is a host file specification. A file specification must be completely contained on a line.

*group-name*

This is a control section that includes specified modules and/or control sections of module(s).

*module-name*

This is the name of a module in the program being linked.

*paddress*

This is a physical address in the form of a hexadecimal number from 0 to FFFFFF.

**ppage**

This is a physical page number in the form of a hexadecimal number from 0 to FFFFF.

**symbol**

This is the name of an external symbol in the program being linked.

**vaddress**

This is a virtual address in the form of a hexadecimal number from 0 to FFFFFFFF ( $2^{32} - 1$ ).

**vpage**

This is a virtual page number in the form of a hexadecimal number from 0 to FFFFF.

Each TLDlnk directive is described below.

**ASSIGN (vpage=ppage{,...})**

The ASSIGN directive causes the specified virtual page to be mapped to the corresponding physical page.

For example,

ASSIGN (40000=C0)

causes the specified virtual page 40000 hex to be mapped to the physical page C0 hex.

ASSIGN (40000=C0, 10000=B0)

causes the specified virtual pages 40000 hex and 10000 hex to be mapped to the physical pages C0 hex and B0 hex, respectively.

**COMMENT [=](")Text to be put in Load Module{"**

The COMMENT directive contains text which TLDlnk puts in the load module. TLDlnk precedes the text within quotes by ";;" to distinguish user inserted comments from those inserted by TLDlnk which begin with ";". All comments specified by COMMENT directives are inserted in the load module immediately following the initial comment which is created by TLDlnk. If

quotes are specified, they must exist at the beginning and end of the text to be treated as a comment.

**DEBUG {file}**

When **DEBUG** is used the linker creates a debug file containing symbols and their values for the symbolic debugger and a traceback file containing call and branching information. If **DEBUG** is not specified, the linker does not produce the debug file and traceback file. The linker puts symbols which were included in the relocatable object file in the debug file and traceback information also in the relocatable object file in the traceback file. The name of the debug and traceback files are derived the same way the map file name is derived as described in the **MAP** switch. The format of the debug and traceback files is described in Appendix A.

This directive has the same functionality as the linker switch **DEBUG** described in Chapter 5.

**END**

This directive is always required (if the End-of-File is not present). It terminates directive input to **TLDlink**, so that any subsequent input is ignored. After this directive is read, **TLDlink** allocates memory and reads the object files to produce the load module.

**FILL (vaddr=vaddress, len=size-in-bytes, {")pattern("})**

The **FILL** directive is used to fill in all unused memory with a user-specifiable value.

The **vaddress** parameter is the starting virtual address of the fill region, the **size-in-bytes** parameter is the number of bytes to be filled with the pattern, and the **pattern** parameter is the pattern used to fill in the fill region. The pair of double quotes ( " ") are required if the fill pattern is a character string.

**GROUP { :group-name= } (name1(, name2...)) ( {attribute(, ...)} ) }**

This directive creates a grouping of control sections. The argument **name** can be **module-name**, **module-name:csect-name**, or **:group-name**. If **module-name** is specified (without **:csect-name**) then the wild card "\*" is assumed for the **csect-name** and all control sections of the specified load module are used. Because the **group-name** is associated with the "null" module, it is always preceded by the null module name: a colon (:). The group name becomes a new control section that includes the

specified control sections and the included control sections may not be specified in any other group. If attributes are specified, then only those control sections with the specified attributes will be included in the group and the group's attributes consist of only those specified in the directive.

This directive, as well as the SET directive, can refer to attributes in pragma Attribute in the source file. Refer to the Reference Document for the TLD Ada Compiler for further information regarding pragma Attribute.

If no data or code attribute is specified and an instruction (code) control section is included in the specification, the group will have the code attribute. If data control section(s) are also specified, a warning message is displayed indicating that the group contains mixed instruction and data control sections and that the code attribute is assumed for the group. If no data or code attribute is specified and no instruction (code) control section is included in the specification, the group will have the data attribute.

The alignment of the group is by the "least common denominator" of all control section alignment values. The length of the group is the sum of the lengths of the included control sections plus necessary alignment. The length (as well as other attributes) of the group may be changed by the SET directive. After all explicit GROUP directives have been applied, the Linker groups any remaining ungrouped control sections and groups by similar attributes. Groups may be used in other group directives.

Attributes may be one or more of the following to select groups with those attributes. The boolean attributes are separated by a comma to denote a logical AND.

**READ**

is a boolean TRUE if the csect is all readable, otherwise, it is FALSE.

**NOREAD**

is a boolean TRUE if the csect is not all readable, otherwise, it is FALSE.

**WRITE**

is a boolean TRUE if the csect is all writable, otherwise, it is FALSE.

**NOWRITE**

is a boolean TRUE if the csect is not all writable, otherwise, it is FALSE.

**CODE**

is a boolean TRUE if the csect is all code, otherwise,  
it is FALSE.

**NOCODE**

is a boolean TRUE if the csect is not all code,  
otherwise, it is FALSE.

**DATA**

is a boolean TRUE if the csect is all data, otherwise,  
it is FALSE.

**NODATA**

is a boolean TRUE if the csect is not all data,  
otherwise, it is FALSE.

To allow grouping of more control sections than can fit in a single directive line, a continuation character can be used or the GROUP directive can be repeated (using the same group name) as many times as needed to include all control sections needed within that group. For example, if the following is in the linker directive file:

```
GROUP :Group_1=(a,b,c)...  
GROUP :Group_1=(d,e,f)...
```

Group\_1 will contain a,b,c,d,e, and f.

Wild card symbols as previously described may be included in the module-name, csect-name, and group-name (which is not the name of the group, but a group to be included).

The ordering of the wild card specifications within the linker directive file is important. If any wild card specification is a subset of another, the subset should be listed first. For example, if the following groups are in the linker directive file:

```
GROUP :Group_1=(abc*:lmn*)...  
GROUP :Group_2=(ab*:lm*)...
```

control section "abcd:lmno" will be included in Group\_1, and since it has been included into a group, will not be included in Group\_2.

The following is an example of incorrect ordering, where the subset is listed after its containing set:

```
GROUP :Group_2=(ab*:lm*)...
```

```
GROUP :Group_1=(abc*:lmn*)...
```

In this example, control section "abcd:lmno" will be included in Group\_2, and since it has been included into a group, will not be included in Group\_1.

```
INCLUDE {()file{...}{}}
```

The INCLUDE directive specifies the file(s) used for subsequent linker input. This is the only linker directive that requires a complete filename (i.e., no file type or extension is appended to the supplied name). If the file name ends in .obj or .olb, the file is assumed to be an object module file. If the file name ends in .lnk, the file is assumed to be a directive file. If only one filespec is specified, the corresponding parentheses are not required. This directive may be repeated.

NOTE: The GROUP and SET directives are used, instead of this directive, to make specific selections of modules and/or control sections to be included in the link.

A directive file may include another directive file. The format of this directive is:

```
INCLUDE filespec{.lnk}
```

where the file extension .lnk may be optionally specified if the file is a directive file, otherwise, if the file is an object module file, its extension must be supplied (i.e., .obj or .olb must be supplied).

An included file may itself include another directive file, that is, in the example above, filespec.lnk may contain yet another directive file. The level of nested directive files allowed depends on the number of files that the operating system permits to be opened simultaneously.

**LET symbol = expression**

When LET is used, the linker sets the specified symbol to the specified value or expression. This directive has the same effect as defining the symbol as an EXPORT in an object module. Any external references to the specified symbol from an object module are set to the value specified in the LET directive. Currently, the expression argument must be a hexadecimal number.

**LIBRARY-**(*()file(...)*)

This directive causes the specified object module library or libraries to be searched to resolve undefined symbols. The parentheses are not required if only one filespec is specified.

The order that the filespecs are specified is the order in which they are searched. If library is used both on the command line and in the directive file, the libraries specified on the command line will be searched first followed by those specified in the directive file.

TLDlnk will process the library directive or switch at the point where it is specified, therefore, it should be specified after includes and searches.

This directive has the same functionality as the linker SEARCH directive and LIBRARY switch which is described in Chapter 5.

**MEMORY** (*mem\_type\_name, base\_address, length\_in\_words, - word\_size\_in\_bits*)

This directive describes a memory unit other than i960 standard memory to which TLDlnk will allocate control sections containing objects specified in pragma Memory\_Unit. The *mem\_type\_name* argument is the character string specified Pragma Memory\_Unit, the *base\_address* argument is the starting address hex value in special memory where the memory unit objects are to be allocated, the *length\_in\_words* argument is the hex value of the size in words of the special memory location, and the *word\_size\_in\_bits* argument is the hex value of the size in bits of each word of special memory.

**RESERVE**(*vaddr=vaddress, len=size\_in\_bytes(...)*)

This directive indicates that no relocatable control sections are to be loaded into the specified address space.

**SEARCH** *file*

When SEARCH is used, TLDlnk searches the specified file for modules which define currently undefined external references. These modules are included as if they had been specified in an INCLUDE directive. Undefined weak external references (i.e., associated with WEAK IMPORT) do not cause inclusion on a search, but if an external is weakly referenced (i.e., associated with WEAK IMPORT) and strongly referenced (i.e., a regular IMPORT), its defining module is loaded by SEARCH. New external references from modules included from the search file may cause additional modules to be included from the search



file, regardless of the order of modules in the search file. For example, if the program references only S, S references T, and the library contains T followed by S, both S and T are included from the library.

This directive has the same functionality as the linker **LIBRARY** directive and **LIBRARY** switch which is described in Chapter 5.

**SET name'({)attribute1=value1(,attribute2=value2,...){})**

This directive sets each specified attribute to the corresponding value for the specified control section or group. The argument name can be module-name, module-name:csect-name, or :group-name. If module-name is specified (without :csect-name) then the wild card "\*" is assumed for the csect-name and all control sections of the specified load module are used. The parentheses are required only if more than one attribute is specified. Because the group-name is associated with the "null" module, it is always preceded by the null module name: a colon (:).

This directive, as well as the **GROUP** directive, can refer to attributes in pragma Attribute in the source file. Refer to the Reference Document for the TLD Ada Compiler for further information on pragma Attribute.

If no data or code attribute is specified and an instruction (code) control section is included in the specification, the control section or group will have the code attribute. If data control section(s) are also specified, a warning message is displayed indicating that mixed instruction and data control sections have been included and that the code attribute is assumed for the group. If no data or code attribute is specified and no instruction (code) control section is included in the specification, the control section or group will have the data attribute.

Wild card symbols may be included in the module-name and csect-name consisting of "\*" which matches one or more characters and "?" which matches exactly one character. All modules and control sections of the object module files listed in the include directive(s) that match the wild card pattern are selected.

Attributes may be one or more of the following to set or reference an attribute value:

**VADDR**

is the beginning virtual address of this csect. It consists of a hex or decimal number. To set address(es) in region 3, an eight-digit, non-negative, hex number must be used.

**PADDR**

is the beginning physical address of this csect. Since the linker does not normally assign physical addresses, this attribute must be set before it is referenced.

**LEN(GTH)**

is the length of this csect.

**ALIGN**

is the alignment used for this csect.

**READ**

is a boolean TRUE if the csect is all readable, otherwise, it is FALSE.

**NOREAD**

is a boolean TRUE if the csect is not all readable, otherwise, it is FALSE.

**WRITE**

is a boolean TRUE if the csect is all writable, otherwise, it is FALSE.

**NOWRITE**

is a boolean TRUE if the csect is not all writable, otherwise, it is FALSE.

**CODE**

is a boolean TRUE if the csect is all code, otherwise, it is FALSE.

**NOCODE**

is a boolean TRUE if the csect is not all code, otherwise, it is FALSE.

**DATA**

is a boolean TRUE if the csect is all data, otherwise, it is FALSE.

**NO DATA**

is a boolean TRUE if the csect is not all data,  
otherwise, it is FALSE.

## APPENDIX C

### APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

.....

type SHORT\_INTEGER is range -32768 .. 32767;

type INTEGER is range -2147483648 .. 2147483647;

type FLOAT is digits 6 range -2.12676E+37 .. 2.12676E+37;

type LONG\_FLOAT is digits 15

range -1.123\_558\_209\_288\_95E+307 .. 1.123\_558\_209\_288\_95E+307;

type DURATION is delta 2.0\*(-14) range -86\_400.0 .. 86\_400.0;

.....

end STANDARD;

## APPENDIX F

The Ada language definition allows for certain machine\_dependencies in a controlled manner. No machine-dependent syntax or semantic extensions or restrictions are allowed. The only allowed implementation-dependencies correspond to implementation-dependent pragmas and attributes, certain machine-dependent conventions, as mentioned in chapter 13 of the MIL-STD-1815A; and certain allowed restrictions on representation clauses.

The full definition of the implementation-dependent characteristics of the TLD VAX/i960 Ada Compiler System is presented in this section extracted from the compiler reference manual.

## **5 i960 TARGET COMPILER**

---

This section identifies correspondences between features of the TLDacs and sections of the Ada Language Reference Manual (LRM).

---

### **5.1 LRM CH.1 - INTRODUCTION**

The formal standards for the Ada Programming Language are provided in the Ada Language Reference Manual (LRM), ANSI/MIL-STD-1815A. TLD Systems has developed TLDacs in the spirit of those standards.

The machine dependencies permitted by the Ada language are identified in LRM Appendix F. No machine dependent syntax, semantic extensions, or restrictions are allowed. The only acceptable implementation dependencies are pragmas and attributes, the machine dependent conventions explained in LRM Chapter 13, and some restrictions on representation clauses.

TLD Systems has developed implementation-dependent software to specifically conform to these restrictions and has developed implementation-independent pragmas and attributes in the spirit of the LRM. This software is described, below, in individual discussions that follow the topical order (within chapters and appendices) of the LRM. For a detailed description of the Run Time environment, refer to the Reference Document for the TLD Ada Run Time System.

---

### **5.2 LRM CH.2 - LEXICAL ELEMENTS**

The items described in this section correspond to the standards in Chapter 2 of the LRM.

The following limits, capacities, and restrictions are imposed by the Ada compiler implementation:

The maximum number of nesting levels for procedures is 10. There is no limit to nesting of ifs, loops, cases, declare blocks, select and accept statements.

The maximum number of lexical elements within a language statement, declaration or pragma is not explicitly limited, but limited depending on the combination of Ada constructs coded.

The maximum number of procedures per compilation unit is 500.

The maximum number of levels of nesting of INCLUDE files is 10. There is no limit on the total number of INCLUDED or WITHed files.

Approximately 2000 user-defined elements are allowed in a compilation unit. The exact limit depends upon the characteristics of the elements.

A maximum of 500 severe (or more serious) diagnostic messages are allowed for a compilation.

The range of status values allowed is the same as the range of integer values, -2147483648..2147483647.

The maximum number of parameters in a procedure call is 20.

The maximum number of characters in a name is 120.

The maximum source line length is 120 characters.

The maximum string literal length is 120 characters.

The source line terminator is determined by the editor used.

Name characters have external representation.

---

## **5.3 LRM CH.3 - DECLARATIONS AND TYPES**

The items described in this section correspond to the standards in Chapter 3 of the LRM.

Number declarations are not assigned addresses and their names are not permitted as a prefix to the 'address attribute.

Objects are allocated by the compiler to occupy one or more 8 bit bytes. Only in the presence of pragma Pack or record representation clauses are objects allocated to less than a word.

'Address can be applied to a constant object to return the address of the constant object.

Except for access objects, uninitialized objects contain an

undefined value. An attempt to reference the value of an uninitialized object is not detected.

The maximum number of enumeration literals of all types is limited only by available symbol table space.

The predefined integer types are:

Integer range -2\_147\_483\_648 .. 2\_147\_483\_647.  
Short\_Integer range -32\_768 .. 32\_768

System.Min\_Int is -2\_147\_483\_648.  
System.Max\_Int is 2\_147\_483\_647.

The predefined real types are:

Float digits 6.  
Long\_Float digits 15.

System.Max\_Digits is 15.

There is no predefined fixed point type name. Fixed point types are implemented as data depending upon the range of values by which the type is constrained.

Index constraints and other address values (e.g., access types) are limited to 2147483647.

The maximum array size is limited to 2147483647.

The maximum string length is 2147483647.

Access objects are implemented as an unsigned integer. The access literal Null is implemented as 0.

There is no limit on the number of dimensions of an array type. Array types are passed as parameters opposite unconstrained formal parameters using a descriptor packet vector.

Additional dimension bounds follow immediately for arrays with more than one dimension.

Packed strings are generated instead of unpacked strings.



---

## **5.4 LRM CH.4 - NAMES AND EXPRESSIONS**

The items described in this section correspond to the standards in Chapter 4 of the LRM.

`Machine_Overflows` is `True`.

`Pragma Controlled` has no effect since garbage collection is never performed.

---

## **5.5 LRM CH.5 - STATEMENTS**

The items described in this section correspond to the standards in Chapter 5 of the LRM.

The maximum number of statements in an Ada source program is undefined and limited only by symbol table space.

Unless they are quite sparse, Case statements are allocated as indexed jump vectors and therefore, are very fast.

Loop statements with a "for" implementation scheme are implemented most efficiently if the range is in reverse and down to zero.

Data declared in block statements is elaborated as part of its containing scope.

---

## **5.6 LRM CH.6 - SUBPROGRAMS**

The items described in this section correspond to the standards in Chapter 6 of the LRM.

Arrays, records, and task types are passed by reference.

---

## **5.7 LRM CH.7 - PACKAGES**

The items described in this section correspond to the standards in Chapter 7 of the LRM.

Package elaboration is performed dynamically, permitting a warm restart without reloading the program.

---

## 5.8 LRM CH.8 - VISIBILITY RULES

Not applicable.

NOTE: TLD has not produced a modification of the item(s) described in this LRM section or documentation parallel to the information in this LRM section.

---

## 5.9 LRM CH.9 - TASKS

The items described in this section correspond to the standards in Chapter 9 of the LRM.

Task objects are implemented as access types pointing to a Process Control Block (PCB).

Type Time in package Calendar is declared as a record containing two integer values: the current value of the real time clock counter and the number of ticks that have elapsed on the countdown timer.

Pragma Priority is supported with a range defined in package System. The restriction on a dynamic expression for a task's priority has been removed consistent with Ada 9X. Note: Like Ada 9X, a pragma Priority placed in the main subprogram remains restricted to a compile time static expression.

Pragma Shared is supported for scalar objects.

TLDada allows either a duration or a clock time to be specified in a delay statement. If a duration is specified, the task is delayed for that duration. If a clock time is specified, the task is delayed until that clock time is reached.

The format for specifying a duration is:

delay expression

where expression is of type Duration.

The format for specifying a clock time is:

delay until expression

where expression is of type Calendar.Time.

Package Calendar is described in the Reference Document for the TLD Run Time System, i960 Target.

---

## **5.10 LRM CH.10 - PROGRAM STRUCTURE/COMPILATION**

Ada Program Library processing is described in the Reference Document for the TLD Ada Library Manager, i960 Target.

Multiple Ada Program Libraries are supported with each library containing an optional ancestor library. The predefined packages are contained in the TLD standard library, I960.LIB

---

## **5.11 LRM CH.11 - EXCEPTIONS**

Exception handling is described in the Reference Document for the TLD Run Time System, i960 Target.

Exception objects are allocated access objects to the exception name string. The implementation of exceptions is described in the Reference Document for the TLD Run Time System, i960 Target.

Exceptions are implemented by the TLD Ada Compiler System to take advantage of the normal policy in real time computer system design to reserve 50% of the duty cycle. By executing a small number of instructions in the prologue of a procedure or block containing an exception handler, a branch may be taken, at the occurrence of an exception, directly to a handler rather than performing the time consuming code of unwinding procedure calls and stack frames. The philosophy taken is that an exception signals an exceptional condition, perhaps a serious one involving recovery or reconfiguration, and that quick response in this situation is more important and worth the small throughput tradeoff in a real time environment.

TLDada allows one task to asynchronously signal a another task by raising an exception in the other task. The following Ada statement may be used in an Ada program to exercise this capability:

```
raise exception_name in task_name
```

There is no direct effect on the task raising the exception. It continues executing the code following the raise statement. The context of the target task is set so that the next time it runs, it will act as if the exception had been raised at the point at which it was last executing. This feature requires the compiler switch XTRA.

---

## **5.12 LRM CH.12 - GENERIC UNITS**

Generic implementation is described in the Reference Document for the TLD Run Time System, i960 Target.

A single generic instance is generated for a generic body, by default. Generic specifications and bodies need not be compiled together nor need a body be compiled prior to the compilation of an instantiation. Because of the single expansion, this implementation of generics tends to be more favorable of space savings. To achieve this tradeoff, the instantiations must, by nature, be more general and are, therefore, somewhat less efficient timewise. Refer to pragma Instantiate for more information on controlling instantiation of a generic.

---

## **5.13 LRM CH.13 - CLAUSES/IMPLEMENTATION**

Representation clause support and restrictions are generally described in Section 5.2.F.

### **Additional Information**

A comprehensive Machine\_Code package is provided and supported. The specification for this package is included in the Machine\_Code\_.Ada file.

The present version of the TLD i960 Ada Compiler System supports two forms of code insertion language features. In addition to the standard LRM form of record aggregate form of code insertions that are fully supported, TLDacs supports an alternative form supplied with package Machine\_Code that defines a procedure for each i960 architecture instruction that is intrinsically implemented inline by using a pragma Interface with a language type of i960; each such procedure results in one instruction being inlined. Because a procedure form is used, the restrictions placed upon the Machine\_Code aggregate form of insertion that prohibit mixing in the same scope with declarations, statements, and functions do not apply. Furthermore, the procedure form offers a more friendly syntax that corresponds more to assembler input that does not require all fields to be specified (as is true for machine code aggregates) and can make use of parameter defaulting for such fields as index registers.

To further support those users who need to write at the assembly level, several additional procedures and pragmas have been added that assist the user in accessing Ada expressions, modifying Ada operands, and in manipulating registers. Pragma Register may be

applied to an Ada object to direct the compiler to allocate (dedicate) the object to the designated register. Use of this object on the left side of an assignment statement will result in the right side expression being computed and loaded into the register associated with the left side object, a la C register variables. Use of the object in a value reference context will result in a use of the value currently found in the associated register. This approach permits direct access to values from complicated Ada expressions, packed and subscripted operands, discriminated record components without having to know how the compiler actually allocated the objects. Two additional procedures are defined, Protect and Unprotect, which each take a register parameter identifying a register that is to be reserved from compiler use within the range of statements bracketed by the Protect/Unprotect call pair.

Pragma Interface with a language type of Interrupt will result in the prologue and epilogue of the indicated procedure generated to conform to the TLDrtx conventions for interrupt handlers. A language type of Void will prevent the compiler from generating any prologue or epilogue code and leave the responsibility for the procedure entry and exit code to the statements within the procedure: usually the above described intrinsically built-in machine code procedures.

Unchecked\_Deallocation and Unchecked\_Conversion are supported.

Procedure Unchecked\_Deallocation (LRM 13.10.1)

Function Unchecked\_Conversion (LRM 13.10.2)

---

## **5.14 LRM CH.14 - INPUT/OUTPUT**

The items described in this section correspond to the standards in Chapter 14 of the LRM.

File I/O operations are not supported.

Input/output packages and associated operations are explained in Section 5.2.F of this manual.

---

## **5.2.A LRM App.A - PREDEFINED LANGUAGE ATTRIBUTES**

The items referenced in this section correspond to the standards in Appendix A of the LRM.

All LRM-defined attributes are supported by the TLDacs.

---

## **5.2.B LRM App.B - PREDEFINED LANGUAGE PRAGMAS**

The items described in this section correspond to the standards in Appendix B of the LRM. Any differences from the implementation described in the LRM are listed below.

### **PRAGMA CONTROLLED**

This pragma is not supported.

### **PRAGMA ELABORATE**

This pragma is implemented as described in the LRM.

### **PRAGMA INLINE**

This pragma is implemented as described in the LRM.

### **PRAGMA INTERFACE**

```
pragma interface (language_name, Ada_entity_name{,string});  
pragma interface (system, Ada_entity_name);
```

Pragma Interface allows references to subprograms and objects that are defined by a foreign module coded in a language other than Ada.

The following interface languages are supported:

- o Assembly for calling Assembly language routines;
- o Intel's 1960 Architecture Specification for defining built-in instruction procedures.

If the `Ada_entity_name` is a subprogram, LRM rules apply to the pragma placement. Pragma Interface may be applied to overloaded subprogram names. In this case, pragma Interface applies to all preceding subprogram declarations if those declarations are not the target of another pragma Interface.

For example:

```
package Test is
  procedure P1;
  pragma Interface (Assembly, P1, "Asm_Routine_1");
  procedure P1 (x:Long_Float);
  pragma Interface (Assembly, P1, "Asm_Routine_2");
end Test;
```

In the example above, the first pragma Interface applies to the first declaration of procedure P1. The second pragma Interface applies to only the second declaration of procedure P1 because the first declaration of P1 has already been the object of a preceding pragma Interface.

If the Ada\_Entity\_Name is an object, the pragma must be placed within the same declarative region as the declaration, after the declaration of the object, and before any reference to the object.

Void may be used as the language\_name to prevent the compiler from generating any prologue or epilogue code and leave the responsibility for the procedure entry and exit code to the statements within the procedure.

If the third parameter is omitted, the Ada name is used as the name of the external entity and the resolution of its address is assumed to be satisfied at link time by a corresponding named entry point in a foreign language module.

If the optional string parameter is present, the external name provided to the linker for address resolution is the contents of the string. Therefore, this string must represent an entry point in another module and must conform to the conventions of the linker being used.

An object designated in an Interface pragma is not allocated any space in the compilation unit containing the pragma. Its allocation and location are assumed to be the responsibility of the defining module.

## **PRAGMA LIST**

```
pragma List (on | off);
```

Compiler switch /LIST must be selected for the pragma List to be effective.

**PRAGMA MEMORY\_SIZE**

`pragma Memory_Size (numeric_literal);`

This pragma is not supported. This number is declared in package System.

**PRAGMA OPTIMIZE**

This pragma is not supported. Compiler switches control compiler optimization.

**PRAGMA PACK**

This pragma is implemented as defined in the LRM.

**PRAGMA PAGE**

This pragma is implemented as defined in the LRM.

**PRAGMA PRIORITY**

This pragma is implemented as defined in the LRM. Priority contains a range defined in System\_Ada.

**PRAGMA SHARED**

This pragma is implemented as defined in the LRM. This pragma may be applied only to scalar objects.

**PRAGMA STORAGE\_UNIT**

`pragma Storage_Unit (numeric_literal);`

This pragma is not supported. This number is declared in package System and has 8 bits per byte.



### **PRAGMA SUPPRESS**

```
pragma Suppress (access_check);  
pragma Suppress (all_checks);
```

The `all_checks` parameter eliminates all run time checks with a single pragma. In addition to the pragma, a compiler switch permits control of run time check suppression by command line option, eliminating the need for source changes.

```
pragma Suppress (discriminant_check);  
pragma Suppress (division_check);  
pragma Suppress (elaboration_check);  
pragma Suppress (index_check);  
pragma Suppress (length_check);  
pragma Suppress (range_check);  
pragma Suppress (overflow_check);  
pragma Suppress (storage_check);
```

### **PRAGMA SYSTEM\_NAME**

```
pragma System_Name (enumeration_literal);
```

This pragma is not supported. Instead, compiler option is used to select the target system and target Ada library for compilation.

---

## **5.2.C LRM APP.C-PREDEFINED LANGUAGE ENVIRONMENT**

The items described in this section correspond to the standards in Appendix C of the LRM.

### **PACKAGE STANDARD**

The specification for this package is included in the `Standard_Ada` file.

---

## **5.2.D LRM APP.D - GLOSSARY**

Not applicable.

---

## **5.2.E LRM APP.E - SYNTAX SUMMARY**

Refer to "Appendix B. Ada Language Syntax Cross Reference" for the TLD cross-referenced expression of this information.

---

## **5.2.F LRM APP.F - IMPLEMENTATION CHARACTERISTICS**

The items described in this section correspond to the standards in Appendix F of the LRM.

### **IMPLEMENTATION-DEPENDENT PRAGMAS**

#### **PRAGMA ADDRESS\_SPACE**

```
pragma Address_Space (name(, subsystem_name));
```

This pragma allows users to specify the association of a compilation unit with a logical address space. This capability will support the definition of i960 Extended Architecture "Domains" and domain calls.

The following switch may be entered on the TLDada command line and used instead of this pragma to associate compilation unit(s) with a logical address space.

```
/address_space=name| (name, subsystem_name)
```

In either the pragma or the switch, *name* is the name of the address space and *subsystem\_name* is the name of the subsystem to which the address space belongs. If *subsystem\_name* is not supplied, then the address space does not belong to a subsystem.

This pragma may appear in any compilation unit. The command line switch may appear in any compilation, and applies to all the compilation units in the compilation.

This capability does not yet allow users to indicate objects that are to be implemented and referenced as independent objects.

TLDlnk will verify that all compilation units in the link have an address space attribute of the same value, or have no address space attribute and will create either a domain (if an address attribute is specified) or a program (if no address space attribute is specified).

## **PRAGMA ADDRESS\_SPACE\_ENTRY**

```
pragma Address_Space_Entry (name{, entry_number}{, entry_type});
```

This pragma allows users to indicate which subprograms represent entries into the defined logical address space. This capability will support the definition of 1960 Extended Architecture "Domains" and domain calls.

The `name` is the name of a previously declared subprogram, the `entry_number` is an integer expression which is evaluatable at compile time, and the `entry_type` is one of the following: Local, Supervisor, or Subsystem. If `entry_type` is not specified, it defaults to Subsystem.

This pragma may appear only in a compilation unit for which an address space has been specified either by pragma or command line switch.

This capability does not yet allow users to indicate objects that are to be implemented and referenced as independent objects.

TLDlink will verify that all compilation units in the link have an address space attribute of the same value, or have no address space attribute and will create either a domain (if an address attribute is specified) or a program (if no address space attribute is specified).

## **PRAGMA ATTRIBUTE**

```
pragma Attribute (Attribute-Name->Attribute-Value, -  
                  Item-Name{,...});
```

This pragma allows grouping of control sections with the specified attribute.

If `Item-Name` is omitted, the specified attribute applies to all control sections in the current module.

If `Item-Name` is `Name'csect`, the specified attribute applies to the control section of the module containing `Name`. `Name` may be a label, procedure, or data object.

If `Item-Name` is `Name'code`, the specified attribute applies to the code control section of the module containing `Name`.

If *Item-Name* is *Name*'data, the specified attribute applies to the data control section of the module containing *Name*.

If *Item-Name* is *Name*'constant, the specified attribute applies to the constant control section of the module containing *Name*.

No other form of *Item-Name* is allowed.

The linker directives GROUP and SET, described in Chapter 4 of the Reference Document for the TLD Linker can refer to attributes in pragma Attribute in the source file.

## **PRAGMA AUDIT**

```
pragma Audit (Ada-name(,...));
```

This pragma causes an error message to be generated for the compilation in which an Ada name, that is specified by this pragma, is referenced. The Ada name may be a package, scope, data, etc.

## **PRAGMA COMPRESS**

```
pragma Compress (subtype_name);
```

This pragma is similar to pragma Pack, but has subtly different effects. Pragma Compress accepts one parameter: the name of the subtype to compress. It is implemented to minimize the storage requirements of subtypes when they are used within structures (arrays and records). Pragma Compress is similar to pragma Pack in that it reduces storage requirements for structures, and its use does not otherwise affect program operation. Pragma Compress differs from pragma Pack in the following ways:

- o Unlike pragma Pack, pragma Compress is applied to the subtypes that are later used within a structure. It is not used on the structures themselves. It only affects structures that later use the subtype; storage in stack frames and global data are unaffected.
- o Pragma Compress is applied to discrete subtypes only. It cannot be used on types.
- o Pragma Compress does not reduce storage to the bit-level. It reduces storage to the nearest "natural machine size". This increases total storage requirements, but minimizes the performance impact for referencing a value.

For example:

```
subtype Small_Int is Integer range 0 .. 255;  
pragma Compress(Small_Int);  
type Num_Array is array (1 .. 1000) of Small_Int;
```

In this example, `Small_Int` will be reduced from a 32-bit object to an unsigned 8-bit object when used in `Num_Array`.

If `pragma Compress` had not been used then `Small_Int` would be the same size as `Integer`. This is because a subtype declaration should not change the underlying object representation. A subtype declaration should only impose tighter constraints on bounds. In this manner a subtype does not incur any extra overhead (other than its range checking), when compared with its base type. `Pragma Compress` is used in those cases where the underlying representation should change for the subtype, therefore:

- o `Small_Int` is compatible with `Integer`. It may be used anywhere an integer is allowed. This includes out and in out parameters to subprograms.
- o A `Small_Int` object is the same size as `Integer` when used by itself. This minimizes run time overhead requirements for single objects allocated in the stack or as global data.
- o `Small_Int` is 8 bits when used within a record or an array. This can dramatically reduce storage requirements for large structures. The access performance for compressed elements is very near that of the un-compressed elements, but a slight performance cost is incurred when the compressed value is passed as an out or in out parameter to a subprogram.

NOTE: `Small_Int`'s storage requirements could be reduced by declaring it as a type rather than a subtype, however, `Small_Int` would not be compatible with `Integer`, and this could cause considerable problems for some users.

## **PRAGMA CONTIGUOUS**

```
pragma Contiguous (type_name | object_name);
```

This pragma is used as a query to determine whether the compiler has allocated the specified type of object in a contiguous block of memory words.

The compiler generates a warning message if the allocation is noncontiguous or is undetermined. The allocation is probably noncontiguous when data structures have dynamically sized components. The allocation is probably undetermined when unresolved private types are forward type declarations.

This pragma provides information to the programmer about the allocation scheme used by the compiler.

## **PRAGMA EXPORT**

```
pragma Export (language_name, ada_entity_name, {string});
```

Pragma Export is a complement to pragma Interface. Export directs the compiler to make the `ada_entity_name` available for reference by a foreign language module. The `language_name` parameter identifies the language in which the module is coded.

Assembly is presently supported by Export. Ada is permitted and presently means the same as Assembly. The semantics of its use is subject to redefinition in future releases of TLDada. Void may be used as the `language_name` to specify the user's language convention. As a result of specifying Void, the Compiler will not allocate local stack space, will not perform a stack check, and will not produce prologue and epilogue code. If the optional third parameter, `string`, is used, the `string` provides the name by which the entity may be referenced by the foreign module. The contents of this `string` must conform to the conventions for the indicated foreign language and the linker being used. TLDada does not make any checks to determine whether these conventions are obeyed.

Pragma Export supports only objects that have a static allocation and subprograms. If the `ada_entity_name` is a subprogram, this Export must be placed in the same scope within the declarative region. If it is an object, the `ada_entity_name` must follow the object declaration.

**NOTE:** The user should be certain that the subprogram and object are elaborated before the reference is made.

#### PRAGMA IF

```
pragma If (compile_time_expression);  
pragma Elsif (compile_time_expression);  
pragma Else;  
pragma End( if);
```

These source directives may be used to enclose conditionally compiled source to enhance program portability and configuration adaptation. These directives may be located where language defined pragmas, statements, or declarations are allowed. The source code following these pragmas is compiled or ignored (similar to the semantics of the corresponding Ada statements), depending upon whether the `compile_time_expression` is true or false, respectively. The primary difference between these directives and the corresponding Ada statements is that the directives may enclose declarations and other pragmas.

NOTE: To use the pragma IF, ELSEIF, ELSE, or END, the /XTRA switch must be used.

#### PRAGMA INCLUDE

```
pragma Include (file_path_name_string);
```

This source directive in the form of a language pragma permits inclusion of another source file in place of the pragma. This pragma may occur any place a language defined pragma, statement, or declaration may occur. This directive is used to facilitate source program portability and configurability. If a partial `file_path_name_string` is provided, the current default pathname is used as a template. A file name must be provided.

NOTE: To use the pragma INCLUDE, the /XTRA switch must be used.

#### PRAGMA INSTANTIATE

```
pragma Instantiate (option(, name));
```

This pragma is used to control instantiation of a particular generic.

To establish a default mode of instantiation for all generic instantiations within the compilation, the following switch may be entered on the TLDada command line and used instead of this pragma:

`/instantiate=option`

In either the pragma or switch, `option` instructs the Compiler to instantiate generics in the manner specified, as described below:

`single_body` - a single body is used for all instantiations

`macro` - each instantiation produces a different body

In this pragma, `name` is the name of the generic to which this pragma applies.

There are two basic forms for this pragma. The first form omits the second parameter, is associated with a generic declaration, and is permitted to occur only within a generic formal part (i.e., after "generic" but before "procedure", "function", or "package"). In this form, the pragma establishes the default mode of instantiation for that particular generic.

The second form uses the second parameter, is associated with the instantiation, and may appear anywhere in a declarative part except within a generic formal part. This form specifies what mode is to be used for the instantiation of the named generic which follows in the scope in which the pragma appears. This form of the pragma takes precedence over the first form.

In the following example, assume the following definition:

```
generic
pragma instantiate(single_body);      -- pragma 1
package G ...
end G;

generic
pragma instantiate(macro);            -- pragma 2
package H ...
end H;
```



```
package A is new G(...);
package B is new G(...);
package C is new H(...);
package D is new H(...);

pragma instantiate(macro, G);           -- pragma 3

package E is new G(...);
package F is new G(...);
```

In the above example, packages A and B share the same body, due to pragma 1. Packages C, D, E, and F will be treated as macro instantiation C and D because macro instantiation is the default for H (due to pragma 2) and for E and F because they follow pragma 3.

In both the pragma and switch:

- o Nested instantiations and nested generics are supported and generics defined in library units are permitted.
- o It is not possible to perform a macro instantiation for a generic whose body has not yet been compiled.

In this pragma:

- o It is also not possible to perform a macro instantiation from inside a single-bodied instantiation, because the macro instantiation requires information at compile time which is only available to a single-bodied generic at execution time.

In the event of a conflict between the pragma and switch, the switch takes precedence.

Please refer to Section 3.12 "Generics" for more information on the advantages and disadvantages in using single\_body generics versus macro generics.

## **PRAGMA INTERFACE\_NAME**

```
pragma Interface_Name (Ada_entity_name, string);
```

This pragma takes a variable or subprogram name and a string to be used by the Linker to reference the variable or subprogram. It has the same effect as the optional third parameter to pragma Interface.

## **PRAGMA IO\_Object**

```
pragma IO_Object (object_name | type_name{,...});
```

An `IO_Object` is an object which is fully contained within a page (4096 bytes) and which begins and ends on a full word (4 byte) boundary. 16 bytes of space preceding the `IO_Object` are reserved by the Compiler for user-specified use. An object is specified to be an `IO_Object` by use of the `pragma IO_Object`.

If `pragma IO_Object` is applied to a type, then any object of that type is an `IO_Object`. If `pragma IO_Object` is applied to an object, then that object is an `IO_Object`.

Only static objects may be `IO_Objects`. If an attempt is made to specify an object which is not static (e.g., an object declared within a subprogram) as an `IO_Object`, TLDada issues a warning and the object is treated as a normal object.

The following is an example of Ada source in which `pragma IO_Object` is applied to an object:

```
...
type Buffer_Type is ...
pragma IO_Object (Buffer_Type);

Buffer_Object : Buffer_Type;
```

An `IO_Object` can only be applied to scalar or composite types and objects but cannot to component(s) of a composite type.

An `IO_Attribute` can be used to determine whether or not an object is an `IO_Object` and to return its value. Refer to section Implementation-Dependent Attributes in this Chapter for more information.

## **PRAGMA INTERRUPT\_KIND**

```
pragma Interrupt_Kind (entry_name, entry_type{, duration});
```

An interrupt entry is treated as an "ordinary" entry in the absence of `pragma Interrupt_Kind`. When `pragma Interrupt_Kind` is used, an interrupt entry may be treated as a "conditional" or "timed" entry.

This pragma must appear in the task specification containing the entry named and after the entry\_name is declared. Three entry\_types are possible: ordinary, timed, and conditional. The optional parameter duration is applicable only to timed entries and is the maximum time to wait for an accept.

For an ordinary entry, if the accept is not ready, the task is queued. For a conditional entry, if the accept is not ready, the interrupt is ignored. For a timed entry, if the accept is not ready, the program waits for the period of time specified by the duration. If the accept is not ready in that period, the interrupt is ignored.

#### **PRAGMA LOAD**

```
pragma Load (literal_string);
```

This pragma makes the Compiler TLDada include a foreign object (identified by the literal\_string) into the link command.

#### **PRAGMA MEMORY\_UNIT**

```
pragma Memory_Unit (mem_type_name, object_name | type_name, {...});
```

TLDacs will locate objects in memory units other than i960 standard memory. Such objects are specified by use of pragma Memory\_Unit. TLDada creates a control section for the specified memory unit and allocates the specified objects or all objects of the specified type to that control section. It passes the memory unit information to TLDlnk in the object module. The user specifies the location and size of the non-standard memory unit to link through a MEMORY directive. The mem\_type\_name is the name of the memory unit and is currently one of the following:

SPE  
BME  
GLOK  
LOT  
SPMMIC  
PBMMIC  
SPMCASIU  
PBMCAIU  
FITS  
MNVM  
WWFROM  
SUBBUS

and either `object_name` (the specified object) or the `type_name` (all objects of that type) may be specified.

For example:

```
pragma Memory_Unit (SPE, Buffer_Type);
```

will collect all objects of `Buffer_Type` in a control section for the memory unit name `SPE`.

The only legal reference to an object in a memory unit is a 'address reference.

## **PRAGMA MONITOR**

```
pragma Monitor;
```

The `pragma Monitor` can reduce tasking context overhead by eliminating context switching. This `pragma` identifies invocation by the compiler. With `pragma Monitor`, a simple procedure call is used to invoke task entry.

Generally, `pragma Monitor` restricts the syntax of an Ada task, limiting the number of operations the task performs and leading to faster execution.

The following restrictions pertain to Ada constructs in monitor tasks:

- o `Pragma Monitor` must be in the task specification.
- o Monitor tasks must be declared in library level, nongeneric packages.
- o A monitor task consists of an infinite loop containing one `select` statement.
- o The "when condition" is not allowed in the `select` alternative of the `select` statement.
- o The only selective wait alternative allowed in the `select` statement is the `accept` alternative.
- o All executable statements of a monitor task must occur within an `accept` body.
- o Only one `accept` body is allowed for each entry declared in the task specification.

If a task body violates restrictions placed on monitor tasks, it is identified as erroneous and the compilation fails.

### **PRAGMA NO\_DEFAULT\_INITIALIZATION**

```
pragma No_Default_Initialization;  
pragma No_Default_Initialization (typename{,... });
```

The LRM requires initialization of certain Ada structures even if no explicit initialization is included in the code. For example, the LRM requires access\_type objects to have an initial value of "NULL." Pragma No\_Default\_Initialization prevents this default initialization.

In addition, initialization declared in a type statement is suppressed by this pragma.

TLD implementation of packed records or records with representation clauses includes default initialization of filler bits, i.e., bits within the allocated size of a variant that are not associated with a record component for the variant. No\_Default\_Initialization prevents this default initialization.

This pragma must be placed in the declarative region of the package, before any declarations that require elaboration code. The pragma remains in effect until the end of the compilation unit.

NOTE: To use the pragma, NO\_DEFAULT\_INITIALIZATION, the /XTRA switch must be used. The use of this pragma may affect the results of record comparisons and assignments.

### **PRAGMA NO\_ELABORATION**

```
pragma no_elaboration;
```

Pragma No\_Elaboration is used to prevent the generation of elaboration code for the containing scope. This pragma must be placed in the declarative region of the affected scope before any declaration that would otherwise produce elaboration code.

This pragma prevents the unnecessary initialization of packages that are initialized by other non-Ada operations. Pragma No\_Elaboration is used to maintain the Ada Run Time Library (TLDrtl).

For example:

```
package Test is
  Pragma No_Elaboration;
  for Program_Status_Word use
    record at mod 8;
    System_Mask at 0*WORD range 0..7;
    Protection_Key at 0*WORD range 10 .. 11; -- bits 8,9 unused
    ...
  end record;
end Test;
```

In the above example, the No\_Elaboration pragma, prevents the generation of elaboration code for package Test since it contains unused bits.

NOTE: To use the pragma, NO\_ELABORATION, the /XTRA switch must be used. The use of this pragma may affect the results of record comparisons and assignments.

#### **PRAGMA NO\_ZERO**

```
pragma No_Zero (record_type_name);
```

If the named record type has "holes" between fields that are normally initialized with zeroes, this pragma will suppress the clearing of the holes. If the named record type has no "holes", this pragma has no effect. When zeroing is disabled, comparisons (equality and non-equality) of the named type are disallowed. The use of this pragma can significantly reduce initialization time for record objects.

#### **PRAGMA PUT**

```
pragma Put (value{, ...});
```

Pragma Put takes any number of arguments and writes their value to standard output at compile time when encountered by the Compiler. The arguments may be expressions of any string, enumeration, integer type, or scalar expression (such as integer'size) whose value is known at compile time. This pragma prints the values on the output device without an ending carriage return; pragma Put\_Line is identical to this pragma, but adds a carriage return after printing all of its arguments.

This pragma is useful in conditionally-compiled code to alert the programmer to problems that might not otherwise come to his attention via an exception or a compile-time error.

This pragma may appear anywhere a pragma is allowed.

### **PRAGMA PUT\_LINE**

```
pragma Put_Line (value{, ...});
```

Pragma Put\_Line takes any number of arguments and writes their value to standard output at compile time when encountered by the Compiler. The arguments may be expressions of any string, enumeration, integer type, or scalar expression (such as integer'size) whose value is known at compile time. This pragma prints the values on the output device and adds a carriage return after printing all of its arguments; pragma Put is identical to this pragma, but prints the values without an ending carriage return.

This pragma is useful in conditionally-compiled code to alert the programmer to problems that might not otherwise come to his attention via an exception or a compile-time error.

This pragma may appear anywhere a pragma is allowed.

### **PRAGMA REGISTER**

```
pragma Register (object_name, register_number);
```

This pragma allows limited register dedication to an object for the purpose of loading registers with complex Ada expressions or storing registers into complex operands within machine code insertion subprograms. The Compiler dedicates the specified register to the specified object until the end of the scope is reached or until it is released through a call to the subroutine, Unprotect, in the Machine\_Code package. The object\_name is the name of the object to be dedicated to the register and register\_number is the register number (without the "R" prefix that is valid for the particular target).

These objects may be used on the left or right side of an assignment statement to load or store the register, respectively.

### **PRAGMA TCB\_EXTENSION**

```
pragma TCB_Extension (value);
```

This pragma is used to extend the size of the Task Control Block on the stack. It can be used only within a task specification. The parameter passed to this program must be static and represents the size to be extended in bytes.

### **PRAGMA UNALIGNED**

```
pragma Unaligned(name, ...);
```

This pragma is used to accommodate an access object that contains, or might contain, an address which is not four byte aligned. The name parameter identifies an access type or object that contains unaligned address values. The name parameter may also refer to a formal parameter passed by address that might be occasionally passed an unaligned actual parameter.

### **PRAGMA WITHIN\_PAGE**

```
pragma Within_Page (type_name);  
pragma Within_Page (object_name);
```

**NOTE:** The *type\_name* or *object\_name* must have been previously declared in the current declaration region and these declarations must be in a static data context (i.e., in a package specification or body that is not nested within any procedure or function).

This pragma instructs the compiler to allocate the specified object, or each object of the specified type, as a contiguous block of memory words that does not span any page boundaries (a page is 4096 bytes).

The compiler generates a warning message if the allocation is noncontiguous or not yet determined (see the description of pragma Contiguous, above). Additionally, the compiler generates a warning message if the pragma is in a nonstatic declarative region. If an object exceeds 4096 bytes, it is allocated with an address at the beginning of a page, but it spans one or more succeeding page boundaries and a warning message is produced.



## **PRAGMA VOLATILE**

```
pragma Volatile (variable_simple_name);
```

This pragma performs the same function as Pragma Shared, however, it also applies to composite types as well as scalar types or access types.

## **IMPLEMENTATION-DEPENDENT ATTRIBUTES**

### **ADDRESS\_TYPE**

The attribute 'Address\_Type is used in a length representation clause to indicate that the address type is to have the characteristics of an access descriptor (with a tagged bit).

The format is:

```
for type-name'Address_Type use Access_Descriptor
```

### **TASK\_ID**

The attribute 'Task\_ID is used only with task objects. This TLD-defined attribute returns the actual system address of the task object.

### **IO\_ATTRIBUTE**

The attribute 'IO\_Attribute is used to determine whether or not an object is an IO\_Object.

When IO\_Attribute is applied to an object, it returns a value of type Object\_Attribute\_Type, which is a private type declared in the package System.

If the object is an IO\_Object, then the value returned is the address of a record containing the address of the object and the number of bits in the object including any bits necessary for padding (and does not include the preceding 16 bytes of reserved user space).

If the object is not an IO\_Object, then the value returned is Invalid\_Object\_Attribute, which is also defined in the package System.

The association of an IO\_Attribute with its IO\_Object is maintained only at compile time. For example, if an IO\_Object is passed as a parameter to a subprogram, then within the subprogram, the IO\_Attribute for that IO\_Object has the value Invalid\_Object\_Attribute.

The following is an example of obtaining the value returned by 'IO\_Attribute for the IO\_Object Buffer\_Object shown in the example above (under the Pragma IO\_Object subsection heading).

A procedure which reads information into an IO\_Object is defined as follows:

```
procedure Get (...; Buff_Attr : System.Object_Attribute_Type; ...);
```

The procedure is called as follows:

```
Get (...; Buffer_Object'IO_Attribute, ...);
```

In the above example, the address of a record containing the address of the object and the number of bits in the object are returned for Buffer\_Object.

## **PACKAGE SYSTEM**

The specification for this package is included in the System\_Ada file.

## **REPRESENTATION CLAUSES**

Record representation clauses are supported to arrange record components within a record. Record components may not be specified to cross a word boundary unless they are arranged to encompass two or more whole words. A record component of type record that has record representation clause applied to it may be allocated only at bit 0. Bits are numbered from right to left with bit 31 indicating the sign bit.

When there are holes (unused bits in a record specification), the compiler initializes the entire record to permit optimum assignment and compares of the record structure. A one-time initialization of these holes is beneficial because it allows block compares and/or assignments to be used throughout the program. If this "optimization" is not performed, record assignments and compares would have to be performed one component at a time, degrading the code.

To avoid this initialization, the user should check to be certain

that no holes are left in the record structure. This may be done by increasing the size of the objects adjacent to the hole or by defining dummy record components that fill the holes. If the latter method is used, any aggregates for the structure must contain values for the holes as well as the "real" components. Even with the extra components, this approach should optimize space and speed in comparison to processing one component at a time.

If the `component_clause` of a record representation specification is not in the same order as the `component_list` of the record specification, the entire record is initialized, as indicated above.

Address clauses are supported for variable objects and designate the virtual address of the object. The Compiler System uses address specification to access objects allocated by non-Ada means and does not handle the clause as a request to allocate the object at the indicated address. Address clauses to specify the address to which code should be relocated, are not supported for subprograms, packages, or tasks.

NOTE: Length clauses are supported for 'Size applied to objects other than task and access type objects and denote the number of bits allocated to the object.

Length clauses are supported for 'Storage\_Size when applied to a task type and denote the number of words of stack to be allocated to the task.

Length clauses are supported for 'Storage\_Size applied to an access type and indicates the number of storage units to be reserved for the collection.

Enumeration representation clauses are supported for value ranges of Integer'First to Integer'Last.

An alignment representation clause has been added that corresponds to Ada 9X that requests a subtype or object to be allocated to an address that is a multiple of the alignment value. Its syntax is

*for object\_or\_subtype'Alignment use expression.*

The alignment expression must be a static value. The use of multiple alignment clauses within the same control section will result in the containing control section assuming an alignment value which is the greatest common multiple (GCM) of the alignment factors occurring within the control section.

## **CONVENTIONS FOR IMPLEMENTATION-GENERATED NAMES DENOTING IMPLEMENTATION-DEPENDENT COMPONENTS**

The Compiler System defines no implementation dependent names for compiler generated record components.

Two naming conventions are used by TLDacs. All visible run time library subprograms and kernel services begin with the character "A\_". Global Run Time System data names begin with the characters "A\$". The unique name created by the compiler for overload resolution is composed of the user name appended with "\$", plus a maximum of three characters derived from the compilation unit name, followed by three digits representing the ordinal of the visible name within the compilation unit. The maximum length of this name is 128 characters.

## **INTERPRETATION FOR EXPRESSIONS APPEARING IN ADDRESS CLAUSES**

Address expression values and objects of type Address represent a location in the program's linear address space.

## **RESTRICTIONS ON UNCHECKED CONVERSIONS**

Unchecked conversion of generic formal private types is not allowed.

## **IMPLEMENTATION-DEPENDENT CHARACTERISTICS OF INPUT-OUTPUT PACKAGES**

**PACKAGE DIRECT\_IO** (LRM 14.2.5)

**PACKAGE IO\_EXCEPTIONS** (LRM 14.5)

**PACKAGE SEQUENTIAL\_IO** (LRM 14.2.3)

Input-Output packages are described in the Reference Document for the TLD Ada i960 Run Time System.

**PACKAGE TEXT\_IO (LRM 14.3.10)**

The following implementation-defined types are declared in Text\_Io:

type Count is integer range 0 .. 511;  
subtype Field is Integer range 0 .. 127;

---

## **6 i960 TARGET COMPILER CHARACTERISTICS**

The characteristics of the target compiler are described in this section.

---

### **6.1 i960 RUN TIME CONVENTIONS**

The Run Time conventions established for the TLD Ada Compiler and Run Time System are explained in the Reference Document for the TLD Run Time System, i960 Target. This information is necessary when the user's application software is coded in a language other than Ada.

---

### **6.2 EXTERNAL NAMES**

External names are supported to a maximum length of 128 characters or the limits imposed by the i960 Linker. The system dependent character, "\_", is left as a "\_" in external names since it is a legal character for the Linker.

---

## **7 RELOCATABLE OBJECT FILES**

TLDada produces Relocatable Object Files containing the results of the compilation.

The TLDada Compiler partitions the generated object module into several separately relocatable control sections. By default, instructions are allocated in control section, \$ISECTS. Literals are allocated in a read-only operand control section, \$CONSS. Statically allocated data are allocated in control section \$DATAS. The NOCSEG switch may be used to combine literals and data into the same control section.

The TLD Relocatable Object File is described in the Reference Document for the TLD 32-Bit Universal Linker, i960 Target.

The control section names and attributes are:

**Relocatable Control Sections**

Name:	\$ISECTS	\$CONSS	\$DATAS	UNMAPPED
	Instructions	Constants	Data	Unmapped
	.....	.....	.....	.....
Attributes:				
INDIRECT	DIRECT	DIRECT	DIRECT	DIRECT
MAPPED	MAPPED	MAPPED	MAPPED	UNMAPPED
RAM/ROM	RAM_OR_ROM	RAM_OR_ROM	RAM	RAM
SU/ROM	MAIN MEMORY	MAIN MEMORY	MAIN_MEMORY	MAIN_MEMORY
EMPTY	NOT EMPTY	NOT EMPTY	NOT EMPTY	NOT EMPTY
DMA	UNPROTECTED	UNPROTECTED	UNPROTECTED	UNPROTECTED
MPRAM	UNPROTECTED	UNPROTECTED	UNPROTECTED	UNPROTECTED
PREG	UNPROTECTED	UNPROTECTED	UNPROTECTED	UNPROTECTED
ROMUNSP	RO,MECH UNSP	RO,MECH UNSP	UNPROTECTED	UNPROTECTED
O/I	INSTRUCT MEM	OPERAND MEM	OPERAND MEM	OPERAND MEM
PRLL	MODULE ALLOC	MODULE ALLOC	MODULE ALLOC	MODULE ALLOC

These attributes are also described in the Reference Document for the TLD 32-Bit Universal Linker, i960 Target. Sections 3 and 4.2 describe TLDlink's use of attributes, Appendix A describes the TLD Relocatable Object File attributes and associated values.

## 8 TARGET REFERENCE TABLE

The following table provides i960 parameter values.

```

-----
-- Purpose:
--% To satisfy the Ada LRM requirement for package SYSTEM
-----

-- type address is range 0..16#FFFF_FFFF#;
type address is range -2_147_483_648..2_147_483_647;
for address'size use 32;

-- an 1960 33-bit access descriptor---we ignore the 33rd bit here
type access_descriptor is range -2_147_483_648..2_147_483_647;
for access_descriptor'size use 32;

type unsigned is range 0..2_147_483_647;
for unsigned'size use 31;

type short_integer is range -32_768..32_767;
for short_integer'size use 16;

type long_integer is range -2_147_483_648..2_147_483_647;
for long_integer'size use 32;

-- Note: The order of the elements in the OPERATING_SYSTEMS and NAME
-- enumerations CANNOT be changed--they must correspond with the values
-- in the CONFIG.CFG file.

type Operating_Systems is (Unix, Netos, Vms, Ucsd, Msdos, Bare, Trump, RTX);

type Name is (Pmachine, M68000, Vax, A1750, Z8002, Z8001,
              Gould, Pdp11, M68000, Pa3200, Caps, Amdahl,
              18086, 180286, 180386, Z80000, M632000, Ibms1,
              M68020, Nebula, Name_X, Hp, Bb1, Hawk, R1666, 1960);

type Object_Attribute_Type is private;
Invalid_Object_Attribute : constant Object_Attribute_Type;

system_name: constant name := 1960;
os_name:     constant operating_systems := RTX;

subtype priority is integer range 1..20; -- 1 is default priority.

--- note: the following priority is probably not valid for the Hawk
---       and will have to be modified when tasking is implemented
subtype interrupt_priority is integer range 1..15;

pragma put_line('>', '>', '>', ' ', system_name,
               ' ', '/', ' ', ' ', os_name, ' ', '<', '<', '<', '<');

-- Language Defined Constants
storage_unit: constant := 8;
memory_size: constant := 16#4000_0000#; -- 256M words per segment
min_int:     constant := -2**31;
max_int:     constant := 2**31-1;
max_digits:  constant := 15;
max_mantissa: constant := 31;
fine_delta:  constant := 2.0**(-31);
ticks_per_second : constant := 1_000_000.0; -- Clock ticks are 1 usecs.
tick            : constant := 1.0/ticks_per_second;
ticks_per_rtc   : constant := 16#100_0000#;

-- system specific constants
address_0:     constant address := 0; -- Zero address
null_address:  constant address := 0; -- Null ptr as system.address
null_AD : constant access_descriptor := 0; -- null AD, untagged

private
type Object_Attribute_Type is record
  Object_Address : Address := null_address;
  Object_Size    : Integer := Integer'first;
end record;
Invalid_Object_Attribute : Constant Object_Attribute_Type :=
  (Object_Address => null_address,
   Object_Size    => Integer'first);
end system;

```

```

-----
--" The following software is the property of TLD Systems, Ltd. "
--" Copyright (C) TLD Systems, Ltd., 1992 "
--" "
--" When this software is delivered to the U.S. Government, "
--" the following applies: "
--" "
--" RESTRICTED RIGHTS LEGEND "
--" Use, duplication, or disclosure by the Government is subject to "
--" restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in "
--" Technical Data and Computer Software clause at 52.227-7013. "
--" TLD Systems, Ltd., Torrance, California "
-----

```

```

-----
-- Source file name:
-- SYSTEM.ADA
--
-- Packages defined:
-- system - system constrained constants and types
--
-- Revision history:
-- 07-21-88 glw new code for Hawk-32:
-- add types: short_integer, long_integer
-- add 'osvs' to operating_systems type
-- add 'hawk' to name type
-- change priority range to 1..200
-- change address range to 0..16#FFFFFFF#
-- change address'size to 32 bits
-- change memory_size to 16#10000000#
-----

```

package system is

```

-----
-- SRS Requirements:
-- #extract requirements
-- #none.
-- #end
-----

```

```

-----
-- Package name:
-- system - system constrained constants and types
--
-- Initialization entry:
-- none
--
-- Types/subtypes defined:
-- address
-- unsigned
-- short_integer
-- long_integer
-- operating_systems
-- name
-- priority
-- interrupt_priority
-- Object_Attribute_Type
--
-- Constants defined:
-- system_name
-- os_name
-- storage_unit
-- memory_size
-- min_int
-- max_int
-- max_digits
-- max_mantissa
-- fine_delta
-- ticks_per_second
-- tick
-- ticks_per_rtc
-- address_0
-- null_address
-- null_AD
-- Invalid_Object_Attribute
-----

```

END  
5-94